Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Bachelorarbeit

Pseudozufallszahlen in der Kryptographie

vorgelegt am: 21. Oktober 2019

Name:	Felix Nils Ortmann
Matrikelnummer:	10004843
Studiengang:	Informatik
E-Mail:	f.ortmann09@t-online.de

Erstprüfer: Prof. Dr. Heribert Vollmer
Prüfer: Dr. Arne Meier
Betreuer: Fabian Müller

Inhaltsverzeichnis

1 Einleitung

Die sichere Verschlüsselung von Nachrichten spielt in der heutigen Zeit eine sehr große Rolle. Für viele Verschlüsselungsverfahren sind zufällige Zeichenketten sehr wichtig. Moderne Computer sind allerdings rein deterministisch, weshalb sie ohne externe Eingaben keine zufälligen Zeichenketten generieren können. Ein Beispiel für eine externe Eingabe ist, dass man die Zeit zwischen zwei Tastenanschlägen misst. Die letzte Stelle der Zeit ist dann ein zufälliges Bit [12, S. 204]. Es ist also relativ aufwendig, ein einzelnes zufälliges Bit zu erhalten. Solche Methoden sind deswegen nur für kurze Zeichenketten geeignet. Verschlüsselungsverfahren benötigen meistens längere Zeichenketten, die z.B. so lang wie die Nachricht sind, die verschickt werden soll.

Deswegen müssen diese kurzen echten zufälligen Zeichenketten verlängert werden. Die verlängerten Zeichenketten sollen die Eigenschaft haben, dass ein möglicher Widersacher diese nicht von einer echten zufälligen Zeichenkette unterscheiden kann. Diese Eigenschaft wird *Pseudozufall* genannt. Falls der Widersacher einige der Bits der Zeichenkette kennt, soll er dadurch nicht die nächsten Bits erraten können. Da Bits nur zwei Werte annehmen können, besteht immer eine Chance von 50%, das nächste Bit richtig zu erraten. Für den Widersacher soll es unmöglich sein, eine deutlich bessere Wahrscheinlichkeit als 50% zu haben, das nächste Bit richtig zu erraten [8].

Das Verlängern von den kurzen Zeichenketten geschieht durch *Bitgeneratoren*. Ein Bitgenerator G ist ein deterministischer Algorithmus, der aus einer Zeichenkette x mit Länge k in Polynomialzeit eine Zeichenkette $G(x)$ der Länge $l(k) > k$ erstellt [12, S. 205]. Damit ein Bitgenerator ein *Pseudozufallszahlengenerator* ist, darf die Wahrscheinlichkeit, dass ein Prädiktor den Next-Bit Test besteht, nicht wesentlich größer als 50% sein. Der Next-Bit Test verläuft so, dass der Prädiktor die ersten i Bits erfragt und für das Bit an der Stelle $i + 1$ einen Wert errät. Wenn der Wert richtig ist, besteht der Prädiktor den Test [8].

Ein Widersacher könnte aber versuchen, mit einem statistischen Test Auffälligkeiten in der erstellten Zeichenkette zu finden. Ein Beispiel für solch einen Test ist es, die Anzahl der Nullen zu zählen. In einer echten zufälligen Zeichenkette hat etwa die Hälfte der Bits den Wert Null. Wenn das in der erstellten Zeichenkette nicht der Fall ist, könnte daran festgestellt werden, dass die Zeichenkette nicht zufällig ist. Des Weiteren reicht es nicht, wenn nur ein paar statistische Tests bestanden werden, da sonst ein Widersacher auf einen anderen Test kommen könnte, der nicht bestanden wird [12, S. 206f.]. Deswegen müssen alle statistischen Tests bestanden werden. Genau das bewies Andrew C. Yao 1982: Ein Bitgenerator ist genau dann ein Pseudozufallszahlengenerator, wenn dieser alle statistischen Tests besteht [13].

Der Blum-Micali Generator ist ein Beispiel für einen Pseudozufallszahlengenerator. Dieser basiert auf dem Prinzip von Hardcore Prädikaten [8]. Ein Prädikat ist wie eine Ja/Nein-Frage, also eine Funktion, die auf 0 oder 1 abbildet. Die Frage nach dem Least Significant Bit einer Zeichenkette ist zum Beispiel ein Prädikat. Um nun zu erklären, was ein Hardcore Prädikat ist, benötigt man Einwegfunktionen. Einwegfunktionen sind Funktionen, die leicht zu berechnen und schwer umzukehren sind. Eine Einwegfunktion

ist zum Beispiel die diskrete Exponentialfunktion mit dem diskreten Logarithmus als Umkehrfunktion [12, S. 127].

Daraus folgt für Hardcore Prädikate: Ein Hardcore Prädikat ist für eine Einwegfunktion f ein Praedikat, das schwer zu berechnen ist, wenn man nur $f(x)$ kennt, aber leicht, wenn man x kennt. Unter Verwendung der diskreten Exponentialfunktion ist die Frage nach dem Most Significant Bit ein Hardcore Prädikat, da die Umkehrfunktion, der diskrete Logarithmus, als schwer zu berechnen gilt [12, S. 209ff.].

Genau dieses Hardcore Prädikat wird für den Blum-Micali Generator verwendet. Dafür wählt man eine zufällige k-Bit Primzahl p, eine Primitivwurzel g und eine zufällige Zahl x zwischen 0 und $p-1$. Anschließen setzt man die Variable $x_1 = x$ und für $i = 2$ bis $i = l(k)$ $x_i = g^{x_{i-1}} \bmod p$. Von allen x_i nimmt man das Most Significant Bit als y_i und gibt als Output die Zeichenkette $y_{l(k)}, \ldots, y_1$ aus [12, S. 212].

Die erzeugten Zeichenketten von Pseudozufallszahlengeneratoren können dann für die sichere Verschlüsselung von Nachrichten verwendet werden. Wir stellen zwei Verfahren vor, bei denen Pseudozufallszahlen helfen können. Zum Einen beim One-Time-Pad und zum Anderen bei der Public-Key Kryptographie.

Beim One-Time Pad wählt Alice eine zufällige Zeichenkette aus, die sie mit Bob geheim teilt. Mit einem Pseudozufallszahlengenerator können beide daraus eine längere Zeichenkette generieren, die so lang wie die Nachricht ist, die Alice an Bob senden möchte. Dann verschlüsselt Alice die Nachricht mit der generierten Zeichenkette und Bob kann die Nachricht mit der gleichen generierten Zeichenkette wieder entschlüsseln. Es müssen also nicht mehr Schlüssel, die so lang wie Nachrichten sind, ausgetauscht werden [12, S. 216].

Bei der Public-Key Kryptographie kann man mithilfe von Pseudozufallszahlen erreichen, dass die gleiche Nachricht nicht das gleichen Kyrptogramm hat. Dadurch werden keine weiteren Informationen über die Nachricht bekannt und sogar das Versenden von 1-Bit Nachrichten ist sicher [12, S. 217].

Des Weiteren gibt es auch Kryptosysteme, die beide Verfahren vereinen, wie z.B. das Blum-Goldwasser Kryptosystem. Dabei wird unter anderem mithilfe des Public-Key eine Pseudozufallszahl generiert, die als One-Time Pad benutzt wird [12, S. 224f.].

Zunächst gibt Kapitel 2 einen Überblick über die mathematischen Grundlagen und einige wichtige Begriffe. In Kapitel 3 zeigen wir die Grundlagen zum Erstellen von Pseuodozufallszahlen und in Kapitel 4 die Konstruktion von Pseuodozufallszahlengeneratoren. Im 5. und 6. Kapitel präsentieren wir die Anwendungen und Vorteile von Pseudozufallszahlen in symmetrischen und Public-Key Kryptosystemen.

2 Grundlagen

Wir definieren zuerst einige wichtigen Begriffe und mathematische Grundlagen, die wir in dieser Arbeit verwenden.

Definition (vernachlässigbar)**.** Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ heißt vernachlässigbar, wenn es für jedes Polynom $p : \mathbb{N} \to \mathbb{N}$ eine ganze Zahl k_0 gibt, sodass

$$f(k) < \frac{1}{p(k)}$$

für $k \geq k_0$ gilt.

Eine vernachlässigbare Funktion ist also kleiner als das Inverse jedes positiven Polynoms. Wir notieren vernachlässigbare Funktionen mit $neg(k)$ [12, S. 126]. Die Abkürzung neg kommt von dem englischen Wort negligible für vernachlässigbar.

In dieser Arbeit geht es oft darum, ob etwas (zum Beispiel eine Funktion) effizient zu berechnen ist. Immer wenn der Begriff effizient vorkommt, bedeutet das, dass es einen Algorithmus geben muss, der diese Funktion in Polynomialzeit berechnet. Bei den Algorithmen gehen wir davon aus, dass diese von einem Computer berechnet werden können. Algorithmen, die wir als nicht effizient bezeichnen, laufen nicht in Polynomialzeit.

Viele der Algorithmen in dieser Arbeit sind probabilistische Algorithmen. Probabilistisch bedeutet, dass im Verlauf des Algorithmus zufällige Zwischenergebnisse gewählt werden, um zu einem Ergebnis zu kommen. Es wird zum Beispiel eine zufällige ganze Zahl gewählt und mit dieser weiter gerechnet. Somit ist der nächste Zustand nicht immer eindeutig definiert, da der nächste Zustand von zufälligen Werten abhängig sein kann [12, S. 68ff.].

Definition (längenerhaltend)**.** Eine Funktion $f\colon \{0,1\}^* \text{ ist längenerhaltend, wenn für jedes } x \in \{0,1\}^* \text{ gilt: } |f(x)| = |x|$ [12, S. 214].

Definition (Permutation)**.** Eine Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$ ist genau dann eine Permutation, wenn jedes $x \in \{0,1\}^*$ ein eigenes Urbild unter f besitzt.

Eine Permutation ist also ein bijektive Funktion von einer Menge auf sich selbst [12, S. 214].

Die folgenden Definitionen und Sätze übernehmen wir aus dem Anhang 3.3 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12].

Definition (Primitivwurzel)**.** Sei $g \in \mathbb{Z}_n^*$, dann ist g eine Primitivwurzel, wenn $ord(g) = \phi(n)$.

$ord(g)$ ist die kleinste Potenz k für die $g^k = 1 \bmod n$ gilt. Diese Potenz wird als Ordnung von g bezeichnet. $\phi(n)$ ist die Anzahl der Elemente in $\mathbb{Z}_n^*$, also $\phi(n) = |\mathbb{Z}_n^*|$. Es lassen sich genau alle Zahlen aus $\mathbb{Z}_n^*$ durch g^i darstellen mit $i = 1, \ldots, \phi(n)$.

Definition (Quadratischer Rest)**.** Eine Zahl $b \in \mathbb{Z}_n^*$ ist ein quadratischer Rest mod n, wenn es ein $x \in \mathbb{Z}_n^*$ gibt, sodass $b = x^2 \bmod n$ gilt.

Satz 2.1 (eulersche Kriterium)**.** *Sei p eine Primzahl größer als 2 und $y \in \mathbb{Z}_p^*$, dann ist y ein quadratischer Rest mod p genau dann, wenn $y^{(p-1)/2} = 1 \; mod \, p$.*

Satz 2.2 (kleiner fermatischer Satz)**.** *Sei p eine Primzahl und $a \in \mathbb{Z}_p^*$, dann gilt*

$$a^{p-1} = 1 \; mod \; p.$$

Satz 2.3. *Sei p eine Primzahl und $a, b \in \mathbb{Z}_p^*$, dann ist ab genau dann ein quadratischer Rest mod p, wenn a und b beide ein quadratischer Rest mod p sind oder a und b beide kein quadratischer Rest mod p sind.*

Satz 2.4 (chinesischer Restsatz)**.** *Sein $n_1, n_2, \ldots, n_k$ paarweise teilerfremd und $N = \prod_{i=1}^{k} n_i$, dann hat das folgende System eine eindeutige Lösung mod N*

$$x = a_1 \; mod \, n_1$$

$$x = a_2 \; mod \, n_2$$

$$\vdots$$

$$x = a_k \; mod \, n_k.$$

Die Lösung ist gegeben durch

$$x = \sum_{i=1}^{k} b_i N_i a_i \; mod \, N,$$

für $N_i = N/n_i$ und $b_i N_i = 1 \; mod \, n_i$.

3 Pseudozufallszahlen

Dem Inhalt dieses Kapitels liegen die Kapitel 10.1 bis 10.3 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

3.1 Pseudozufall

Da ein Computer keine echten Zufallszahlen generieren kann, führen wir für computergenerierte Zufallszahlen den Begriff *Pseudozufall* ein. Informell bedeutet Pseudozufall, dass eine computergenerierte Zeichenkette nicht von einer echten zufälligen Zeichenkette unterscheidbar ist. Genauer gesagt soll ein möglicher Gegenspieler nicht effizient herausfinden können, dass die Zeichenkette nicht echt zufällig ist. Darüber hinaus soll es dem Gegenspieler nicht möglich sein, die einzelnen Stellen der Zeichenkette effizient zu berechnen.

Damit ein Computer Pseudozufallszahlen generieren kann, benötigt dieser eine externe Eingabe von einer Zeichenkette. Diese werden wir ab jetzt als *Saat* bezeichnen. Ein möglicher Weg, die Saat einzugeben ist, die Zeit zwischen zwei Tastenanschlägen auf einer Tastatur zu messen. Von der gemessenen Zeit nimmt man die letzte Ziffer als eine Ziffer für die Saat. Dieser Vorgang muss mehrmals wiederholt werden, um eine längere Zeichenkette zu bekommen, und ist deswegen sehr zeitaufwendig. Für lange zufällige Zeichenketten ist dieses Verfahren daher nicht geeignet. Wir benötigen also eine Methode um längere Zeichenketten aus einer kurzen Saat zu generieren. Ein Computer generiert diese mithilfe eines *Bitgenerators*.

3.2 Bitgeneratoren

Definition (Bitgenerator)**.** Ein *Bitgenerator* G ist ein deterministischer Algorithmus, der in Polynomialzeit aus einer Zeichenkette $x \in \{0,1\}^k$ eine längere Zeichenkette $G(x)$ mit der Länge $l(k) > k$ generiert.

Ein Bitgenerator bekommt eine Saat als Eingabe und gibt eine Zeichenkette aus, die mindestens ein Bit länger als die Saat ist. Da der Algorithmus eine polynomielle Laufzeit besitzt, ist jedes einzelne Bit b_k der Ausgabezeichenkette effizient zu berechnen.

Wir wollen nun, dass die von Bitgeneratoren generierten Zeichenketten pseudozufällig sind. Dafür müssen die generierten Bits $b_1, b_2, b_3, \ldots$ die oben genannte Eigenschaft von pseudozufälligen Zeichenketten vorweisen, dass ein möglicher Gegenspieler die generierten Bits nicht effizient berechnen kann, ohne die Saat zu kennen.
Um diese Eigenschaft genauer festzulegen, definieren wir zuerst den Gegenspieler genauer. Wir bezeichnen ihn als *Prädiktor*.

Definition (Prädiktor)**.** Ein *Prädiktor* P ist ein Polynomialzeialgorithmus, der als Eingabe die ersten s Bits einer von einem Bitgenerator generierten Zeichenkette $G(x)$ bekommt und eine Aussage über das nächste Bit $s+1$ ausgibt.

Für alle Prädiktoren soll es unmöglich sein, das Bit an der Stelle $s+1$ mit einer größeren Wahrscheinlichkeit als $\frac{1}{2}+neg(k)$ richtig zu bestimmen, wenn den Prädiktoren die ersten s Bits $b_1, \ldots, b_s$ der Ausgabezeichenkette $G(x)$ bekannt sind. Der Ursprung der $\frac{1}{2}$ ist, dass ein Prädiktor das Bit zufällig, zum Beispiel durch einen Münzwurf, bestimmen kann. In dem Fall entspricht die Wahrscheinlichkeit $\frac{1}{2}$, das Bit an der Stelle s richtig zu bestimmen.

Diese Bedingung definieren wir genauer mit einem *Next-Bit Test*, den jeder Prädiktor in der Hälfte aller Fälle nicht bestehen soll.

3.3 Next-Bit Test

Next-Bit Test

1: Eingabe: Die von einem Bitgenerator G generierte Zeichenkette $G(x) = y_1, y_2, \ldots, y_{l(k)}$.
2: $i \leftarrow 1$
3: **while** $i \leq l(k)$ **do**
4: P fragt entweder nach dem nächsten Bit y_i oder gibt eine Vermutung b aus.
5: **if** P gibt Vermutung aus **then**
6: der Test ist vorbei und P besteht wenn $b = y_i$
7: **else**
8: P bekommt das nächste Bit y_i
9: $i \leftarrow i + 1$
10: **end if**
11: **end while**
12: P besteht den Test nicht, weil keine Vermutung gemacht wurde

Der Prädiktor erfragt nacheinander die ersten i Bits und gibt für das Bit an der Stelle $i+1$ eine Vermutung ab. Wenn der Prädiktor alle Bits erfragt ohne eine Vermutung abzugeben oder die Vermutung falsch ist, hat der Prädiktor den Next-Bit Test nicht bestanden. Der Test wir vom Prädiktor bestanden, falls dieser das Bit and der Stelle $i+1$ richtig bestimmt.

Da der Prädiktor das nächste Bit maximal mit einer Wahrscheinlichkeit von $\frac{1}{2}+neg(k)$ richtig bestimmen soll, folgt für den Next-Bit Test die Wahrscheinlichkeit:

$$\Pr[P \text{ besteht den Next-Bit Test}] \leq \frac{1}{2} + neg(k)$$

Denn wenn der Prädiktor P eine Vermutung ausgibt, ist die Wahrscheinlichkeit, dass diese richtig ist und er somit den Test besteht, maximal $\frac{1}{2}+neg(k)$. Wenn der Prädiktor keine Vermutung ausgibt, hat dieser den Test nicht bestanden.

Die Bits der Ausgabezeichenkette eines Bitgenerators sind resultierend daraus nicht effizient berechenbar, wenn alle Prädiktoren den Next-Bit Test maximal mit einer Wahrscheinlichkeit von $\frac{1}{2} + neg(k)$ bestehen.

Somit können wir den Next-Bit Test als ein Kriterium für Bitgeneratoren benutzen, die pseudozufällige Zeichenketten generieren. Diese Bitgeneratoren nennen wir auch Pseudozufallszahlengeneratoren.

Definition (Pseudozufallszahlengenerator)**.** Ein Bitgenerator G ist genau dann ein Pseudozufallszahlengenerator, wenn für jeden Prädiktor P die Wahrscheinlichkeit, dass P den Next-Bit Test besteht, höchstens $\frac{1}{2} + neg(k)$ ist.

Der Next-Bit Test hilft zu zeigen, dass die einzelnen Bits der Ausgabezeichenkette von einem Prädiktor nicht effizient berechnet werden können. Wir wollen aber zusätzlich, dass die Ausgabezeichenkette nicht von einer echten zufällige Zeichenkette unterscheidbar ist.

Deswegen werden wir die Definition von Pseuodzufallszahlengeneratoren mithilfe von statistischen Tests erweitern.

3.4 Statistische Tests

Informell sind statistische Tests Algorithmen, die anhand einer bestimmten Statistik einer generierten Zeichenkette eine Aussage darüber treffen, ob es sich um eine echte zufällige Zeichenkette handelt oder nicht. Das verdeutlichen wir an dem folgenden Beispiel.

Beispiel 3.1. Ein Beispiel für einen statistischen Test ist ein Algorithmus, welcher die Anzahl der Nullen in einer Zeichenkette zählt. In einer echten zufälligen Zeichenkette entspricht die Anzahl der Nullen im Durchschnitt ungefähr 50% der Länge der Zeichenkette. Wenn der Anteil der Nullen in einer Pseudozufallszahl deutlich von den 50% abweicht, könnte erkannt werden, dass die Zahl nicht echt zufällig ist. Der Algorithmus gibt dann eine andere Ausgabe aus, als bei einer echten zufälligen Zeichenkette.

Ein Pseudozufallszahlengenerator muss alle statistischen Tests bestehen, damit die Ununterscheidbarkeit der generierten Zeichenketten zu echten zufälligen Zeichenketten gewährleistet ist. Es reicht nicht aus, dass nur eine große Menge an statistischen Tests bestanden wird. Ansonsten könnte ein anderer statistischer Test gefunden werden, der relevante Informationen über die Zeichenkette offenbart.

Im Folgenden definieren wir, was es bedeutet, einen statistischen Test zu bestehen. Die Definition haben wir aus [8] abgeleitet.

Definition (statistischer Test)**.** Sein p und l Polynome, dann ist ein statistischer Test eine Menge von Schaltkreisen $C = \{C_k\}$ mit weniger als $p(k)$ Gattern, genau $l(k)$ Eingabebits und einem Ausgabebit.

Ein Schaltkreis C_k erhält seine Eingabe zum Beispiel von einem Bitgenerator G mit einer generierten Zeichenkette $G(x)$ der Länge $l(k)$.

Definition. Sei $\Pr^C_{k,G}$ die Wahrscheinlichkeit, dass C_k nach der Eingabe einer von einem Bitgenerator generierten Zeichenkette eine 1 ausgibt, und $\Pr^C_{k,Z}$ die Wahrscheinlichkeit,

dass C_k nach der Eingabe einer zufällige Zeichenkette der Länge $l(k)$ eine 1 ausgibt, dann besteht der Bitgenerator G alle statistischen Tests C, wenn gilt:

$$|\mathrm{Pr}_{k,G}^{C} - \mathrm{Pr}_{k,Z}^{C}| \leq neg(k)$$

Ein statistischer Test C muss bei pseudozufälligen Zeichenketten auf die gleiche Weise antworten, wie bei einer echten zufälligen Zeichenkette der gleichen Länge. Die Wahrscheinlichkeit, dass eine bestimmte Ausgabe ausgegeben wird, muss gleich sein.

Wir wollen nun einen Zusammenhang zwischen Pseudozufallszahlengeneratoren und statistischen Testen herstellen. Der Informatiker Andrew C. Yao lieferte 1982 dazu das folgende wichtige Ergebnis.

Satz 3.1 (Yao 1982 [13])**.** *Ein Bitgenerator ist genau dann ein Pseudozufallszahlengenerator, wenn dieser alle statistischen Tests besteht.*

Beweis. An dieser Stelle liefern wir nur eine Beweisskizze. Die Idee ist folgende: Wir wollen zeigen, dass ein Prädiktor einen statistischen Test, den ein Bitgenerator nicht besteht, zum Bestehen des Next-Bit Tests verwenden kann. Der Prädiktor könnten diesen statistischen Test bei einer Zeichenkette durchführen und anhand der Ausgabe das nächste Bit mit einer Wahrscheinlichkeit, die größer als $\frac{1}{2} + neg(k)$ ist, richtig bestimmen. Dann würde der Bitgenerator auch den Next-Bit Test nicht bestehen und somit wäre der Bitgenerator per Definition kein Pseudozufallszahlengenerator. Ein Pseudozufallszahlengenerator besteht deswegen alle statistischen Tests.

Wenn ein Bitgenerator alle statistischen Tests besteht, besteht dieser den Next-Bit Test, da der Next-Bit Test nach unserer Definition auch ein statistischer Test ist. Deswegen ist ein Bitgenerator der alle statistischen Tests Besteht, ein Pseudozufallszahlengenerator. □

3.5 Einwegfunktionen

In diesem Abschnitt führen wir Einwegfunktionen ein. Diese sind aufgrund ihrer Eigenschaften sehr wichtig für die Konstruktion von Pseudozufallszahlengeneratoren. Dem Inhalt dieses Abschnitts und der beiden folgenden Abschnitte liegen die Kapitel 6.1 und 6.2 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

Definition (Einwegfunktion)**.** Eine Einwegfunktion ist eine Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$ mit den Eigenschaften:

(1) f ist effizient zu berechnen.

(2) f ist schwer umzukehren. Das heiß jeder probabilistische Algorithmus, der $f(x)$ umkehren soll, wenn er eine zufällige Zahl $y = f(x)$ als Eingabe gegeben hat, besitzt eine vernachlässigbare Wahrscheinlichkeit, x richtig zu berechnen.

Ein Beispiel für eine vermutliche Einwegfunktion ist die diskrete Exponentialfunktion dexp.

Definition (Diskrete Exponentialfunktion)**.** Sei p eine Primzahl, g eine Primitivwurzel mod p und $x \in \mathbb{Z}_p^*$. Dann gilt:

$$\mathrm{dexp}(p, g, x) = g^x \bmod p.$$

Die diskrete Exponentialfunktion ist effizient zu berechnen, da Exponentiation mod p in Polynomialzeit durchgeführt werden kann. Ein möglicher Algortihmus dafür ist die binäre Exponentiation mit „square and multiply“.

Das Inverse der diskreten Exponentialfunktion ist die diskrete Logarithmusfunktion.

Definition (Diskrete Logarithmusfunktion)**.** Sei p eine Primzahl, g eine Primitivwurzel mod p und $y = g^x mod\, p$. Dann gilt:

$$\mathrm{dlog}(p, g, y) = x.$$

Die diskrete Logarithmusfunktion gilt als nicht effizient zu berechnen. Wir nehmen an, dass kein effizienter Algorithmus bekannt ist. Einer der schnellsten Algorithmen ist der Zählkörpersieb-Algorithmus. Dieser hat die Laufzeit $O(exp(c(log(p))^{\frac{1}{2}}(log(log(p)))^{\frac{2}{3}}))$, wobei c eine Konstante ist. Die Berechnung der diskreten Logarithmusfunktion wird auch als das diskreter Logarithmus-Problem bezeichnet.

Da ein Pseuodozufallszahlengeneratoren (der Blum-Micali Generator), den wir in dieser Arbeit konstruieren wollen, ohne die nicht effiziente Berechenbarkeit der diskreten Logarithmusfunktion nicht funktionieren würden, werden wir im folgenden Abschnitt eine möglichst genaue Annahme dazu definieren.

Ein möglicher Widersacher soll nur eine vernachlässigbare Wahrscheinlichkeit haben, eine zufällige Instanz des diskreter Logarithmus-Problems korrekt zu lösen.

3.6 Die Diskreter Logarithmus Annahme

Definition (Diskreter Logarithmus Annahme)**.** Für jeden probabilistischen Polynomialzeit-Algorithmus A gilt mit genügend großem k:

$$\Pr[A(p, g, y) = \mathrm{dlog}(p, g, y)] < neg(k).$$

p ist eine zufällige k-Bit Primzahl, g ist eine zufällige Primitivwurzel mod p und y ist eine zufällige Zahl aus $\mathbb{Z}_p^*$.

Jeder probabilistische Algorithmus A zur Berechnung der diskreten Logarithmusfunktion hat nur eine vernachlässigbare Wahrscheinlichkeit, das richtige Ergebnis zu bestimmen.

Diese Annahme bedeutet, dass das diskreter Logarithmus-Problem so gut wie immer nicht effizient berechenbar ist. Das ist auch wichtig, da nur eine kleine Teilmenge, für die das Problem effizient zu lösen ist, reichen würde, um das Problem allgemein effizient zu lösen.

Satz 3.2. *Sei A ein Polynomialzeitalgorithmus der für jede k-Bit Primzahl p und jede Primitivwurzel g mod p das diskreter Logarithmus-Problem für eine Teilmenge $B_p \subseteq \mathbb{Z}_p^*$ löst, mit $|B_p| \geq \varepsilon \cdot |\mathbb{Z}_p^*|$ für $\varepsilon > 0$. Dann gibt es einen probabilistischen Algorithmus der das diskreter Logarithmus-Problem allgemein für $\mathbb{Z}_p^*$ in polynomieller Laufzeit in Abhängigkeit von k und $1/\epsilon$ löst.*

Der Beweis ist aus [12, Seite 128].

Beweis. Sei A der oben gegebene Polynomialzeit-Algorithmus. Wir benutzen den folgenden Algorithmus:

Algorithmus B

```
1: Eingabe:(p, g, y) p ist eine Primzahl, g eine Primitivwurzel mod p und y ∈ Z_p^*.
2: while true do
3:     Wähle ein zufälliges c ∈ Z_p^*
4:     z ← g^c mod p
5:     w ← A(p, g, yz mod p)
6:     if g^w = yz mod p then
7:         return: w − c
8:     end if
9: end while
```

Es wird immer wieder eine zufällige Zahl c gewählt, bis der Algorithmus A $\operatorname{dlog}(p, g, yz \bmod p)$ erfolgreich berechnet. Dann impliziert $z = g^c \bmod p$ (Zeile 4)

$$y = \frac{yz}{z} = \frac{g^w}{g^c} \bmod p = g^{w-c} \bmod p$$

Deswegen gibt der Algorithmus B $\operatorname{dlog}(p, g, y) = w - c$ aus.

Nun müssen wir abschätzen, wie oft die While-Schleife durchlaufen werden muss, um Erfolg zu haben.

Der Algorithmus B hat genau dann Erfolg, wenn yz in der Teilmenge B_p liegt. Denn dann kann A $\operatorname{dlog}(p, g, yz \bmod p)$ erfolgreich berechnen.

Da g eine Primitivwurzel modulo p ist, wird jedes Element von $\mathbb{Z}_p^*$ gleich wahrscheinlich getroffen, wenn c zufällig gewählt wird. Daher entspricht die Wahrscheinlichkeit, dass yz für ein zufälliges c zu B_p gehört,

$$\frac{|B_p|}{|\mathbb{Z}_p^*|} \geq \epsilon.$$

Sei $\delta = \frac{|B_p|}{|\mathbb{Z}_p^*|}$ genau diese Wahrscheinlichkeit. Dann ist δ auch die Wahrscheinlichkeit, dass der gesamte Algorithmus in einem bestimmten Schleifendurchlauf eine Ausgabe tätigt. Deswegen ist die erwartete Anzahl der erforderlichen Durchläufe $\frac{1}{\delta}$. Die Anzahl der erwarteten Durchläufe $\frac{1}{\delta}$ ist nach oben durch $\frac{1}{\epsilon}$ beschränkt, weil $\delta \geq \epsilon$ gilt.

Jede Zeile von B lässt sich in Polynomialzeit durchführen. Das gilt vor allem auch für Zeile 5, da wir angenommen haben, dass A ein Polynomialzeitalgorithmus ist. Die

erwartete Anzahl an Schleifendurchläufen ist kleiner als $\frac{1}{\epsilon}$. Deswegen hat B auch eine polynomielle Laufzeit in Abhängigkeit von k und $\frac{1}{\epsilon}$. □

Satz 3.3. *Unter der diskreter Logarithmus Annahme ist die Funktion dexp eine Einwegfunktion.*

Beweis. Dafür schauen wir uns die Definition einer Einwegfunktion an.
Die Eigenschaft (1) ist erfüllt, da dexp in Polynomialzeit berechenbar ist.
(2) entspricht der diskreter Logarithmus Annahme, wenn man f durch *dexp* ersetzt. □

3.7 Faktorisierung Annahme

Eine weitere wichtige Annahme, die wie treffen, um Einwegfunktionen zu finden, ist, dass die Faktorisierung des Produkts von zwei zufällige k-Bit Primzahlen nicht effizient berechnet werden kann. Für zwei Primzahlen p, q kann das Produkt pq nicht effizient wieder in die beiden Faktoren p und q zerlegt werden.

Definition (Faktorisierung Annahme)**.** Für jeden probabilistischen Polynomialzeit-Algorithmus A gilt mit ausreichend großem k:

$$\Pr[A(n) = (p, q)] \leq neg(k).$$

$n = pq$ und p, q sind k-Bit Primzahlen.

Wir nehmen also an dass jeder probabilistischen Polynomialzeit-Algorithmus A nur eine vernachlässigbare Wahrscheinlichkeit hat, bei der Eingabe einer Zahl n, die Primfaktoren dieser Zahl berechnen zu können.

Der Zahlkörpersieb Algorithmus gilt auch für die Faktorisierung als einer der schnellsten Algorithmen. Dessen Laufzeit ist aber nicht polynomiell.

3.8 Hardcore Prädikate

Wenn wir eine Einwegfunktion f auf eine Zeichenkette $x \in \{0,1\}^*$ anwenden, dann ist die gesamte Zeichenkette $f(x)$ per Annahme nicht effizient umzukehren. Das gilt aber nicht für alle Bits von x. Deswegen führen wir Hardcore Prädikate ein. Mit diesen können wir einzelne Bits generieren, die aus $f(x)$ nicht effizient berechenbar sind.

Definition (Prädikat)**.** Ein Prädikat ist ein Funktion $B\colon \{0,1\}^* \to \{0,1\}$.

Ein Prädikat ist vergleichbar mit einer Frage, die nur zwei mögliche Antworten hat, zu einer Zeichenkette, da die Ausgabe nur ein Bit ist. Das Least Significant Bit ist zum Beispiel ein Prädikat oder die Summe aller Bits mod 2.

Definition (Hardcore Prädikat)**.** Ein Hardcore Prädikat zu einer Einwegfunktion f ist ein Prädikat, welches nicht effizient zu berechnen ist, wenn nur $f(x)$ bekannt ist, und effizient zu berechnen ist, wenn x bekannt ist.

Wir berechnen zum Beispiel von einer Zeichenkette x den Funktionswert $f(x)$ mit der Einwegfunktion f. Dann stellt wir eine Frage nach dem Most Significant Bit von x. Diese Frage sollen ein Gegenspieler nur beantworten können, wenn er x kennt und nicht beantworten können, wenn er nur $f(x)$ wir.

Ein Gegenspieler könnte versuchen, die Antwort der Frage zu erraten, wenn ihm nur $f(x)$ bekannt ist, indem er zufällig eine der beiden möglichen Antworten wählt. Dabei beträgt die Wahrscheinlichkeit, dass der Widersacher die Frage richtig beantwortet $\frac{1}{2}$. Ein Hardcore Prädikat soll dementsprechend die folgende zwei Eigenschaften besitzen.

Ein Prädiakt $B\colon \{0,1\}^* \to \{0,1\}$ ist ein Hardcore Prädikat von einer Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$ genau dann wenn

- es einen probabilistischen Polynomialzeit-Algorithmus zum Berechnen von $B(x)$ gegeben x gibt.
- Für jeden probabilistischen Polynomialzeit-Algorithmus A und $x \in \{0,1\}^k$ gilt

$$\Pr[A(f(x)) = B(x)] \leq \frac{1}{2} + neg(k).$$

Satz 3.4. *Wenn $B\colon \{0,1\}^* \to \{0,1\}$ ein Hardcore Prädikat zu einer Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$ ist, dann gilt für $x \in \{0,1\}^k$*

$$|Pr[B(x) = 0] - Pr[B(x) = 1]| \leq neg(k).$$

Ein Hardcore Prädikat ist also unvoreingenommen bei einer zufälligen Eingabe. Die Antwortmöglichkeiten 0 und 1 sind annähernd gleich wahrscheinlich. Dieser Satz ist wichtig, da wir mit Hardcore Prädikaten Pseuodozufallszahlengeneratoren konstruieren wollen. Denn dafür müssen 0 und 1 gleich wahrscheinlich sein, damit alle statistischen Tests bestanden werden. Der Beweis zu Satz 3.4 ist aus [12, Seite 128].

Beweis. Wir nehmen an, dass das Resultat nicht gilt. Dann gilt für alle $x \in \{0,1\}^k$

$$(1)\ |\Pr[B(x) = 0] - \Pr[B(x) = 1]| > neg(k).$$

Außerdem gilt

$$(2)\ \Pr[B(x) = 0] + \Pr[B(x) = 1] = 1.$$

Wir formen (2) nach $\Pr[B(x) = 1]$ um und setzen die Gleichung in (1) ein

$$|2 \cdot \Pr[B(x) = 0] - 1| > neg(k).$$

Wir addieren 1 auf beiden Seiten und dividieren anschließend durch 2

$$\Pr[B(x) = 0] > \frac{1}{2} + \frac{1}{2} \cdot neg(k).$$

Diese Wahrscheinlichkeit entspricht genau der Wahrscheinlichkeit von dem Polynomialzeitalgorithmus A, der bei Eingabe $f(x)$ immer 0 ausgibt

$$\Pr[A(f(x)) = B(x)] = \Pr[B(x) = 0] > \frac{1}{2} + \frac{1}{2} \cdot neg(k).$$

A hat eine Wahrscheinlich, die mehr als vernachlässigbar größer als $\frac{1}{2}$ ist, $B(x)$ korrekt zu berechnen. Das widerspricht der Tatsache, dass $B(x)$ ein Hardcore Prädikat ist. □

Wir wollen nun ein konkretes Beispiel für ein Hardcore Prädikat durchgehen. Dafür benutzen wir die Einwegfunktion dexp.

Zur Wiederholung: Sei p eine Primzahl, g eine Primitivwurzel mod p und $x \in \mathbb{Z}_p^*$, dann gilt

$$\text{dexp}(p, g, x) = g^x \bmod p.$$

Zwei offensichtliche Prädikate sind das Least und Most Significant Bit von x. Dafür definieren wir uns zwei entsprechende Funktionen.

$$\text{least}(x) = \begin{cases} 0, & \text{wenn } x \text{ gerade ist} \\ 1, & \text{sonst} \end{cases}$$

$$\text{most}(x) = \begin{cases} 0, & \text{wenn } x < (p-1)/2 \\ 1, & \text{sont} \end{cases}$$

Die Zahl x hat hierbei die gleiche Anzahl an Bits wie p und wird vorne mit Nullen aufgefüllt, sodass die Anzahl der Bits von x und p übereinstimmt. Außerdem ist das Prädikat most(x) nicht genau das Most Significant Bit von x. Es beschreibt einen Bereich, für den alle Werte von x, die in diesem Bereich liegen, das Most Significant Bit 0 haben, egal welchen Wert p hat. Es kann also sein, dass das Most Significant Bit einer Zeichenkette 0 ist und most eine 1 ausgibt. Wenn most eine 0 ausgibt, ist das Most Significant Bit auch 0.

Im folgenden Abschnitt werden wir sehen, dass least(x) kein Hardcore Prädikat von dexp ist. Das Prädikat most(x) ist aufgrund der diskreter Logarithmus Annahmen ein Hardcore Prädikat.

Um diese beiden Aussagen zu beweisen, benötigen wir zunächst ein wenig Zahlentheorie.

Wir definieren uns die Menge der quadratischen Reste mod n für $x \in \mathbb{Z}_n^*$ als

$$Q_n = \{b \in \mathbb{Z}_n^* | b = x^2 \bmod n\}.$$

Außerdem benötigen wir das eulersche Kriterium(Grundlagen Satz 2.1) und den folgende Satz. Dadurch können wir die Berechnung so vereinfachen, dass wir nur eine einfache Exponentiation mod p durchführen müssen, um das Least Significant Bit zu berechnen.

Satz 3.5. *Sei p eine Primzahl und $b \in Q_p$, dann gibt es einen probabilistischen Polynomialzeitalgorithmus, der die zwei Quadratwurzeln von b mod p berechnet.*

Beweis. Die 2 ist die einzige gerade Primzahl und für diesen Fall ist die Aussage trivial. Für ungerade Primzahlen müssen wir zwei Fälle unterscheiden.

Der erste Fall ist, wenn $p = 4m + 3$ für $m \in \mathbb{Z}^+$. Um die beiden Quadratwurzeln zu bestimmen, wollen wir $b \bmod p$ so umformen, dass b als das Produkt der beiden Quadratwurzeln dargestellt wird. Da b ein quadratischer Rest mod p ist, können wir das eulersche Kriterium anwenden. Nach dem eulerschen Kriterium gilt $b^{(p-1)/2} = 1 \bmod p$. Daraus folgt

$$b^{(p-1)/2+1} = b^{(p-1)/2} \cdot b = b \bmod p.$$

In die Gleichung setzen wir den Wert von p ein und vereinfachen diese danach

$$b^{(p-1)/2+1} = b^{(4m+2)/2+1} = b^{2m+2} = (\pm b^{m+1})^2 \bmod p$$

Die zwei Quadratwurzeln von b sind also $\pm b^{m+1} \bmod p$. Die Berechnung von b^{m+1} lässt sich in Polynomialzeit durchführen. Daher gibt es eine Polynomialzeitalgorithmus, wenn $p = 4m + 3$.

Der zweite Fall ist $p = 4m + 1$ für $m \in \mathbb{N}$. Der Beweis für Primzahlen dieser Form ist deutlich umfangreicher. Da das Resultat zum Beispiel in [3] ausführlich bewiesen wurde und der Beweis den Rahmen dieser sprengen würde, lassen wir ihn hier weg. □

Nun können wir beweisen, dass least(x) kein Hardcore Prädikat von dexp(p, g, x) ist. Dafür müssen wir zeigen, dass es einen Polynomialzeit-Algorithmus gibt, der least(x) berechnet.

LEAST

```
1: Eingabe: (p, g, b), p ist Primzahl, g eine Primitivwurzel mod p und b = g^x mod p
2: c ← b^((p−1)/2) mod p.
3: if c = 1 then
4:     return 0
5: else
6:     return 1
7: end if
```

Wir müssen zeigen, dass LEAST das Prädikat least(x) richtig berechnet und eine polynomielle Laufzeit hat. Dafür darf $c = 1$ nur genau dann gelten, wenn least$(x) = 0$.

Wenn least$(x) = 0$, dann ist x eine gerade Zahl und wir können x als $2k$ schreiben. Daraus folgt $b = g^x = (g^k)^2 \bmod p$. Somit ist b ein quadratischer Rest mod p und wir können das eulersche Kriterium (Grundlagen Satz 2.1) verwenden. Mit dem eulerschen Kriterium folgt $c = 1$ in Zeile 2.

Nun zeigen wir die andere Richtung. Wenn $c = 1$, dann ist b nach dem eulerschen Kriterium ein quadratischer Rest mod p. Somit gilt $b = g^x = (g^y)^2 = g^{2y} \bmod p$ für $0 \leq y < p - 1$. Das y liegt zwischen 1 und $p - 2$, da x nur Werte zwischen 1 und $p - 1$ annehmen kann und y kleiner als x sein muss. Da g eine Primitivwurzel mod p ist, ist

$x = 2y \bmod p-1$. Das können wir auch als $x = 2y + t \cdot (p-1)$ schreiben für $t \in \mathbb{Z}$. Dann ist x eine Summe von zwei geraden Zahlen und somit ist x auch gerade und $\text{least}(x) = 0$.

Der Algorithmus gibt also $\text{least}(x)$ korrekt aus, wenn $\text{dexp}(p, g, x)$ gegeben ist, und hat eine polynomielle Laufzeit. $\text{least}(x)$ ist somit kein Hardcore Prädikat von dexp.

Nun wollen wir zeigen dass $\text{most}(x)$ ein Hardcore Prädikat von dexp ist. Dafür wollen wir zeigen dass die diskreter Logarithmus Annahme nicht gilt, wenn es einen Polynomialzeit-Algorithmus MOST gibt, der $\text{most}(x)$ berechnet, wenn $\text{dexp}(p, g, x)$ gegeben ist.

Dies erreichen wir, indem wir einen effizienten Algorithmus D konstruieren, welcher mithilfe von MOST das diskreter Logarithmus Problem löst. Dadurch würde der Algorithmus D die diskreter Logarithmus Annahme widerlegen.

Die Idee für dem Algorithmus D ist, dass wir von b immer wieder die Quadratwurzel berechnen und dadurch x in jedem Durchgang nach rechts verschieben. In jedem Durchgang geben wir das nächste Bit, als Least Significant Bit vom verschobenen x aus.

Algorithmus D

1: Eingabe: (p, g, b), p ist eine Primzahl, g eine Primitivwurzel mod p und $b = g^x \bmod p$.
2: $i \leftarrow 0$, $x_0 \leftarrow 0$
3: **while** $b \neq 1$ **do**
4: $\quad c \leftarrow \text{LEAST}(p, g, b)$
5: $\quad$ **if** c $= 1$ **then**
6: $\quad\quad b \leftarrow \frac{b}{g}$ und $x_i \leftarrow 1$
7: $\quad$ **else**
8: $\quad\quad x_i = 0$
9: $\quad$ **end if**
10: $\quad r_1, r_2 \leftarrow \sqrt{b} \bmod p$
11: $\quad$ **if** $\text{MOST}(p, g, r_1) = 0$ **then**
12: $\quad\quad b \leftarrow r_1$
13: $\quad$ **else**
14: $\quad\quad b \leftarrow r_2$
15: $\quad$ **end if**
16: $\quad i \leftarrow i + 1$
17: **end while**
18: **return** $x \leftarrow x_i \dots x_1 x_0$.

Mit dem LEAST Algorithmus wird zuerst das Least Significant Bit von x berechnet (Zeile 4). Wenn dieses 1 ist, teilen wir b durch g und speichern das aktuelle Least Significant Bit 1 in x_i (Zeile 6). Dann hat b den Wert $b = \frac{g^x}{g} = g^{x-1} \bmod p$. Weil x vorher ungerade war, ist der Exponent von g jetzt gerade und somit ein quadratischer Rest mod p. Das bedeutet, dass wir in Polynomialzeit die beiden Quadratwurzeln r_1, r_2 von $b \bmod p$ berechnen können (Satz 3.4). Wenn x schon gerade ist, speichern wir das Least Significant Bit 0 in x_i (Zeile 8) und der Algorithmus kann direkt die Quadratwurzeln berechnen. Wenn x ungerade ist, kann der Algorithmus problemlos mit $b = g^{x-1}$ wei-

ter rechnen, da für uns das Least Significant Bit nicht mehr wichtig ist, wenn wir im nächsten Schritt x nach rechts verschieben wollen.

Wir verschieben x nach rechts, indem wir die Quadratwurzel von b berechnen(Zeile 10). Eine Wurzel ist $r_1 = g^{x/2}$ und die andere $r_2 = g^{x/2+(p-1)/2}$. r_2 ist eine Wurzel von b, da $b = g^x = g^{x+(p-1)} \bmod p$ gilt. Nach unserer Definition für das Prädikat most(x) gibt dieses für alle Werte von x kleiner als $(p-1)/2$ eine 0 und für alle anderen Werte eine 1 aus. Der Algorithmus MOST gibt für die beiden Wurzeln auf jeden Fall verschiedene Werte aus und kann daher zwischen den beiden Wurzeln unterscheiden. Weil wir x nach rechts verschieben wollen, wählt der Algorithmus D die Wurzel, für die MOST 0 ausgibt(Zeile 11-15). Die Wurzel ist dann unser neuer Wert für b mit dem wir weiter rechnen.

Am Ende der While-Schleife erhöht der Algorithmus i um 1. Dann wird der Vorgang mit dem neuen Wert für b wiederholt. Der Algorithmus berechnet in jedem Schleifendurchlauf das nächste Bit bis $b = 1$. Am Ende gibt der Algorithmus x komplett aus.

Jede Zeile des Algorithmus ist in Polynomialzeit berechenbar und die Anzahl der Schleifendurchläufe ist durch die Länge von x beschränkt. Der Algorithmus D ist also ein effizienter Algorithmus zur Berechnung von $\mathrm{dlog}(p, g, b)$. Das bedeutet, dass die diskreter Logarithmus Annahme nicht mehr gelten würde.
Daraus geht das folgende Ergebnis hervor.

Satz 3.6 (Blum und Micali [8])**.** *Unter der diskreter Logarithmus Annahme ist most(x) ein Hardcore Prädikat von dexp.*

4 Konstruktion von Pseudozufallszahlengeneratoren

In diesem Abschnitt zeigen wir, wie wie konkret einen Pseudozufallszahlengenerator konstruieren können. Das zeigen wir zuerst an dem Beispiel des Blum-Micali Generators. Dem Inhalt dieses Kapitels liegt das Kapitel 10.4 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

4.1 Blum-Micali Generator

Der Blum-Micali Generator ist ein Bitgenerator, benannt nach den beiden Informatikern Manuel Blum und Silvio Micali. Dieser verwendet das Hardcore Prädikat $\text{most}(x)$ von $\text{dexp}(p, g, x)$ aus dem letzten Abschnitt. Der Generator verlängert eine zufällige k-Bit lange Saat x zu einer $l(k)$-Bit langen pseudozufälligen Zeichenkette, wobei l ein Polynom ist. Die Funktionsweise ist die folgende:

Blum-Micali Generator

1: Eingabe: Eine zufällige k-Bit Primzahl p, eine Primitivwurzel g mod p und $x \in \mathbb{Z}_p^*$.
2: Setze $x_1 = x$ und berechne $x_i = g^{x_{i-1}} \bmod p$ für $i = 2$ bis $l(k)$.
3: Berechne $b_i = \text{most}(x_i)$
4: **return** Die Zeichenkette $b_{l(k)}, b_{l(k)-1}, \ldots, b_2, b_1$.

Der Blum-Micali Generator berechnet also nacheinander alle Werte x_i mit der diskreten Exponentialfunktion. Dann werden mit dem Hardcore Prädikat $\text{most}(x)$ Bits generiert, die nicht effizient berechnet werden können.

Satz 4.1 (BLum und Micali [8])**.** *Unter der diskreter Logarithmus Annahme ist der Blum-Micali Generator ein Pseudozufallszahlengenerator.*

Der Beweis liegt dem Beweis von Blum und Micali aus [8] für Satz 4.1 zugrunde.

Beweis. Wir nehmen an, dass diese Aussage nicht stimmt und somit der Blum-Micali Generator kein Pseudozufallszahlengenerator ist. Dann folgt mit Satz 3.1, dass der Generator den Next-Bit Test nicht besteht. Es gibt dann einen Prädiktor P, der das Bit an der Stelle $j+1$ mit einer Wahrscheinlichkeit bestimmen kann, die größer als $\frac{1}{2} + neg(k)$ ist, wenn P die vorherigen j Bits kennt.

Wir können nun zeigen, dass wir mit einem solchen Prädiktor einen Algorithmus erstellen können, welcher $\text{most}(x)$ von $\text{dexp}(p, g, x)$ mit einer Wahrscheinlichkeit, die größer als $\frac{1}{2} + neg(k)$ ist, richtig berechnet. Das ist ein Widerspruch zu Satz 3.6, dass $\text{most}(x)$ ein Hardcore Prädikat von $\text{dexp}(p, g, x)$ ist.

Der folgende Polynomialzeitalgorithmus A kann $\text{most}(x)$ mit einer größeren Wahrscheinlichkeit als $\frac{1}{2} + neg(k)$ bestimmen, wenn es so einen Prädiktor P gibt. Sei $n = l(k)$ die Länge der generierten Zeichenkette und $x_0 = x$ ist unbekannt.

Algorithmus A

1: Eingabe: $x_2 = g^{x_1} \bmod p$.
2: Generiere die Zeichenkette $(b_j + 1, \ldots, b_2) = (\text{most}(x_{j+1}), \ldots, \text{most}(x_2))$.
3: Verwende den Prädiktor P mit der Eingabe $(b_j + 1, \ldots, b_2)$.
4: **return** Die Ausgabe von P.

Unser Ziel ist es, dass P most(x) richtig bestimmt. Dafür bestimmt der Algorithmus zuerst $x_3, \ldots, x_n$. Das jeweils nächste x wird mit $x_i = g^{x-1} \bmod p$ berechnet. Aus der Sequenz $x_2, \ldots, x_n$ erstellt der Algorithmus dann die Zeichenkette $b_n, \ldots, b_2$ mit $b_i = \text{most}(x_i)$. Das Bit, das P bestimmen soll, ist b_1 und liegt somit an der Stelle n der Zeichenkette. Die Eingabe für den Prädiktor P wäre dann $n - 1$ Bits lang.

Für den Prädiktor P benötigen wir eine Eingabe der Länge j, da dieser das Bit an der Stelle $j+1$ bestimmt. Wir wollen also die Zeichenkette so verschieben, dass b_1 an der Stelle $j + 1$ liegt. Deswegen übergibt der Algorithmus A dem Prädiktor P die Zeichenkette $(b_j + 1, \ldots, b_2) = (\text{most}(x_j + 1), \ldots, \text{most}(x_2))$. Diese Zeichenkette entspricht den ersten j-Bits der Zeichenkette, die wir generieren, wenn wir statt mit x_1 mit $\text{dlog}^{n-j-1}(p, g, x_1)$ als Saat anfangen. Da wir mit dexp jeweils das nächste x_{i+1} berechnen und dlog eine Bijektion ist, berechnen wir mit dlog jeweils das vorherige x_{i-1}. $\text{dlog}^{n-j-1}(p, g, x_1)$ entspricht somit x_{2+j-n}. Das letzte x der Sequenz ist dann x_{1+j}, da die Zeichenkette insgesamt die Länge n hat. x_1 ist somit an der Stelle $j+1$ von hinten betrachtet. Da der Blum-Micali Generator die Hardcore Prädikate von rechts nach links ausgibt, ist dann auch b_1 an der Stelle $j + 1$.

Dem Prädiktor P übergibt der Algorithmus die Zeichenkette $(b_j + 1, \ldots, b_2)$. Wir müssen nun zeigen, dass dieser das Bit b_1 an der Stelle $j+1$ mit einer Wahrscheinlichkeit berechnen kann, die größer als $\frac{1}{2} + neg(k)$ ist. Am Anfang dieses Beweises haben wir angenommen, dass P genau das kann.

Die Laufzeit von A ist polynomiell, da dexp in Polynomialzeit berechnet werden kann und die Anzahl der Berechnungen von dexp durch $l(k)$ beschränkt ist. Das Berechnen der Hardcore Prädikate ist auch in Polynomialzeit möglich und der Prädiktor P hat per Definition eine polynomielle Laufzeit.

Mit dem Polynomialzeitalgorithmus A können wir also $b_1 = \text{most}(x)$ mit einer Wahrscheinlichkeit, die größer als $\frac{1}{2} + neg(k)$ ist, berechnen. Das ist ein Widerspruch dazu, dass most(x) ein Hardcore Prädikat von $\text{dexp}(p, g, x)$ ist. □

Der Blum-Micali Generator hat das Problem, dass die Bits in umgekehrter Reihenfolge ausgegeben werden. Man muss also vorher wissen wie viele Bits man benötigt und kann die generierte Zeichenkette nicht einfach verlängern, da neue Bits vorne hinzugefügt werden. Das Problem können wir einfach beheben, indem wir die Reihenfolge ändern, in der der Generator die Bits ausgibt.

Satz 4.2. *Sei G ein Pseudozufallszahlengenerator und $\overleftarrow{G}$ der Bit Generator, der die Ausgabe von G nimmt und in umgekehrter Reihenfolge ausgibt, dann ist $\overleftarrow{G}$ auch ein Pseudozufallszahlengenerator.*

Beweis. Die Idee für diesen Beweis ist, zu zeigen, dass G kein Pseudozufallszahlengenerator ist, wenn $\overleftarrow{G}$ keiner ist. Dafür benutzen wir die Aussage von Satz 3.1, dass ein Bitgenerator genau dann ein Pseudozufallszahlengenerator ist, wenn dieser alle statistischen Tests besteht. Das bedeutet, dass es einen statistischen Test T gibt, den $\overleftarrow{G}$ nicht besteht, wenn $\overleftarrow{G}$ kein Pseudozufallszahlengenerator ist.

Sei $\overleftarrow{T}$ ein statistischer Test, der die Reihenfolge der Eingabezeichenkette umdreht und sich danach wie T verhält. Diesen Test besteht G nicht und ist somit kein Pseudozufallszahlengenerator. Dies ist ein Widerspruch zu unser ursrpünglichen Annahme. □

4.2 Allgemeine Konstruktion von Pseudozufallszahlengeneratoren

Wir wollen nun zeigen, wie man allgemein einen Pseudozufallszahlengenerator erstellen kann, wenn man eine Einwegfunktion kennt. Der Blum-Micali Generator funktioniert nur, wenn die diskreter Logarihtmus Annahme gilt. Wenn in der Zukunft der diskrete Logarithmus effizient berechnet werden kann, benötigen wir eine andere Einwegfunktion, um Pseudozufallszahlengeneratoren zu konstruieren.

Mit dem folgenden Ergebnis von Oded Goldreich und Leonid A. Levin von 1989 können wir zu jeder Einwegfunktion eine Einwegfunktion mit Hardcore Prädikat bestimmen.

Satz 4.3 (Goldreich und Levin [9])**.** *Sei f eine Einwegfunktion und $g(x,r) = (f(x), r)$, wobei $|x| = |r| = k$, dann ist*

$$B(x,r) = \sum_{i=1}^{k} x_i r_1 \, mod \, 2.$$

ein Hardcore Prädikat von der Einwegfunktion g.

x und r sind Zeichenketten mit der gleichen Länge und x_i und r_i stehen für das Bit an der Stelle i der jeweiligen Zeichenkette. Die Zeichenkette r wird als Padding für die Funktion g benutzt. g hängt an den Funktionswert $f(x)$ die Zeichenkette r an. Dann ist die Berechnung des Skalarprodukts mod 2 der binären Zeilenvektoren x und r ein Hardcore Prädikat.

Beweis. Siehe [9]. □

Nun zeigen wir, wie wir allgemein einen Pseudozufallszahlengenerator konstruieren können.

Satz 4.4. *Sei $f : \{0,1\}^* \to \{0,1\}^*$ eine längenerhaltende Einwegpermutation mit dem Hardcore Prädikat $B : \{0,1\}^* \to \{0,1\}$, dann ist*

$$G : \{0,1\}^k \to \{0,1\}^{k+1},\ G(x) = (f(x), B(x))$$

ein Pseudozufallszahlengenerator.

Beweis. Die Idee für diesen Beweis ist die folgende. Wir wollen zeigen, dass G alle statistischen Tests besteht und somit ein Pseudozufallszahlengenerator ist(Satz 3.1), indem wir das Gegenteil annehmen. Das bedeutet, dass es einen statistischen Test T gibt, welcher $G(x) = (f(x), B(x))$ von einer echter zufälligen Zeichenkette $z \in \{0,1\}^{k+1}$ unterscheiden kann. Da f eine längenerhaltende Permutation ist, ist $f(x)$ gleichverteilt über $\{0,1\}^k$. Die Ausgabe von T sagt uns also nur etwas über das Bit $B(x)$ aus. Mit diesem Test lässt sich dann ein Algorithmus erstellen, der $B(x)$ mit einer größeren Wahrscheinlichkeit als $\frac{1}{2} + neg(k)$ bestimmt. $B(x)$ wäre dann kein Hardcore Prädikat von f.

Wir nehmen nun an, dass gilt

$$(1)\ \Pr[T(G(x)) = 1] - \Pr[T(z) = 1] \geq neg(k),$$

wobei $x \in \{0,1\}^k$ und $z \in \{0,1\}^{k+1}$. T gibt also eher eine 1 aus, falls T $G(x)$ als Eingabe hat, anstatt eine zufälligen Eingabe z der gleichen Länge.

Mit dem Test T erstellen wir den folgenden Polynomialzeitalgorithmus A zum bestimmen von $B(x)$.

Algorithmus A

```
1: Eingabe: f(x)
2: b ∈ {0,1}
3: c ← T(f(x), b)
4: if c = 1 then
5:     return b
6: else
7:     return b̄
8: end if
```

Die Wahrscheinlichkeit, dass A das Hardcore Prädikat $B(x)$ richtig bestimmt, können wir mit bedingten Wahrscheinlichkeiten berechnen.

Definition (bedingte Wahrscheinlichkeit)**.** Die bedingte Wahrscheinlichkeit von einem Ereignis A unter der Voraussetzung von Ereignis B ist definiert als

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

für $\Pr[B] > 0$.

Der Algorithmus A kann $B(x)$ in zwei möglich Fällen korrekt bestimmen:

Der erste Fall ist, wenn $b = B(x)$ und $c = 1$ gilt. Dann gibt A $b = B(x)$ richtig aus. Die Wahrscheinlichkeit für diesen Fall schreiben wir $\Pr[c = 1 \cap b = B(x)]$.

Der zweite Fall ist, wenn $b \neq B(x)$ und $c = 0$ gilt. Dann gibt A $\bar{b} = B(x)$ richtig aus. Die Wahrscheinlichkeit für diesen Fall schreiben wir $\Pr[c = 0 \cap b \neq B(x)]$.

Die Wahrscheinlichkeit, dass A das Hardcore Prädikat $B(x)$ richtig berechnet, ist die Summe dieser Wahrscheinlichkeiten

$$\Pr[A(f(x)) = B(x)] = \Pr[c = 1 \cap b = B(x)] + \Pr[c = 0 \cap b \neq B(x)].$$

Wir können die Gleichung für bedingte Wahrscheinlichkeiten nach $\Pr[A \cap B]$ umstellen. Dann erhalten wir die Gleichung

$$\Pr[A \cap B] = \Pr[A|B] \cdot \Pr[B].$$

Daraus folgt

$$\begin{aligned}(2)\ \Pr[A(f(x)) = B(x)] = &\Pr[b = B(x)] \cdot \Pr[c = 1|b = B(x)] \\ &+\Pr[b \neq B(x)] \cdot \Pr[c = 0|b \neq B(x)]\end{aligned}$$

Es gilt $\Pr[b = B(x)] = \Pr[b \neq B(x)] = \frac{1}{2}$, da b zufällig gewählt wird und $B(x)$ mit der gleichen Wahrscheinlichkeit 0 oder 1 ausgibt (Satz 3.4).

Aus der Ungleichung (1) können wir die gesuchte bedingte Wahrscheinlichkeit für den ersten Fall herleiten. Es gilt

$$\Pr[T(G(x)) = 1] = \Pr[T(f(x), B(x)) = 1] = \Pr[c = 1|b = B(x)].$$

Wenn wir die Ungleichung (1) nach dieser Wahrscheinlichkeit umformen, erhalten wir

$$(3)\ \Pr[c = 1|b = B(x)] \geq \Pr[T(z) = 1] + neg(k).$$

Für die bedingte Wahrscheinlich für den zweiten Fall müssen wir zuerst $\Pr[T(z) = 1]$ genauer definieren. Die Wahrscheinlichkeit, dass T für eine zufällige Eingabe z eine 1 ausgibt, entspricht der Wahrscheinlichkeit $\Pr[T(f(x), b) = 1]$, da b zufällig gewählt wird.

$$\Pr[T(z) = 1] = \Pr[T(f(x), b) = 1] = \Pr[c = 1 \cap b = B(x)] + \Pr[c = 1 \cap b \neq B(x)].$$

Mit bedingter Wahrscheinlichkeit folgt

$$\begin{aligned}\Pr[T(z) = 1] = &\Pr[b = B(x)] \cdot \Pr[c = 1|b = B(x)] \\ &+\Pr[b \neq B(x)] \cdot \Pr[c = 1|b \neq B(x)]\end{aligned}$$

$\Pr[c = 1|b \neq B(x)]$ ist die Gegenwahrscheinlichkeit von $\Pr[c = 0|b \neq B(x)]$. Wir müssen also die Gleichung für $\Pr[T(z) = 1]$ nach $\Pr[c = 1|b \neq B(x)]$ umformen

$$\Pr[c = 1|b \neq B(x)] = 2 \cdot \Pr[T(z) = 1] - \Pr[c = 1|b = B(x)],$$

und dann die Gegenwahrscheinlichkeit bestimmen

$$(4)\ \Pr[c = 0|b \neq B(x)] = 1 - 2 \cdot \Pr[T(z) = 1] + \Pr[c = 1|b = B(x)].$$

Wir setzen nun zuerst die Gleichung (4) in die Gleichung (2) ein

$$\Pr[A(f(x)) = B(x)] = \frac{1}{2} \cdot (2 \cdot \Pr[c = 1 | b = B(x)] \\ +1 - 2 \cdot \Pr[T(z) = 1])$$

und danach die Ungleichung (3)

$$\Pr[A(f(x)) = B(x)] \geq \frac{1}{2} + neg(k).$$

Der Algorithmus A kann $B(x)$ mit einer Wahrscheinlichkeit, die größer als $\frac{1}{2} + neg(k)$ ist, bestimmen. Das ist ein Widerspruch dazu, dass $B(x)$ ein Hardcore Prädikat ist. □

Der Pseudozufallszahlengenerator aus Satz 4.4 ist in der Realität eher nicht zu gebrauchen, da die Eingabezeichenkette nur um ein Bit erweitert wird. In den meisten Anwendungsfällen von Pseudozufallszahlengeneratoren werden deutlich Zeichenketten benötigt.

Wir wollen nun zeigen, dass wir unter den gleichen Voraussetzungen, wie für Satz 4.4, einen Pseudozufallszahlengenerator konstruieren können, der eine Eingabezeichenkette der Länge k zu einer Zeichenkette der Länge $l(k) > k$ verlängert, für jedes Polynom l.

Satz 4.5 (Blum und Micali [8]). *Sei* $f\colon \{0,1\}^* \to \{0,1\}^*$ *eine längenerhaltende Einwegpermutation mit dem Hardcore Prädikat* $B\colon \{0,1\}^* \to \{0,1\}$. *Wenn* $l(\cdot)$ *ein Polynom ist, ist*

$$G\colon \{0,1\}^k \to \{0,1\}^{l(k)}$$

definiert durch

$$G(x) = (B(x), B(f(x)), B(f^2(x)), \ldots, B(f^{l(k)-1}(x)))$$

ein Pseudozufallszahlengenerator.

Wir lassen den Beweis für diesen Satz weg, da wir den Satz schon für ein konkretes Beispiel in Satz 4.1 bewiesen haben. Wenn wir für den Generator G aus Satz 4.5 die $\mathrm{dexp}(p, g, x)$ als längenerhaltende Einwegpermutation und $\mathrm{most}(x)$ als Hardcore Prädikat wählen, erhalten wir den Blum-Micali Generator.

5 Symmetrische Verschlüsselung

Für dieses und die folgenden Kapitel führen wir die folgenden Charaktere ein, die jeweils eine bestimmte Rolle einnehmen.

Alice und Bob sind die Hauptcharaktere, die eine Nachricht M untereinander austauschen wollen. In unserem Fall sendet immer Alice eine Nachricht an Bob.

Eve ist die Gegenspielerin, die den Austausch zwischen Alice und Bob abhört und versucht aus der verschlüsselten Nachricht, dem Kryptrogramm C, Informationen über die Nachricht zu erlangen.

Dem Inhalt dieses Kapitels liegt das Kapitel 5 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

5.1 Symmetrische Verschlüsselungsverfahren

Ein symmetrisches Verschlüsselungsverfahren ist ein Kryptosystem, bei dem die Verschlüsselung und die Entschlüsselung mit dem selben Schlüssel K durchgeführt werden. Alice und Bob haben einen gemeinsamen Schlüssel. Formal definieren wir ein symmetrisches Kryptosystem wie folgt.

Definition (symmetrisches Kryptosystem)**.** Ein symmetrische Kryposystem ist ein 5-Tupel$(\mathcal{M}, \mathcal{K}, \mathcal{C}, e(\cdot,\cdot), d(\cdot,\cdot))$, wobei $\mathcal{M}$ die Menge der möglich Nachrichten , $\mathcal{K}$ die Menge der möglichen Schlüssel und $\mathcal{C}$ die Menge der möglichen Kryptogramme ist. Die Verschlüsselungsfunktion ist

$$e\colon \mathcal{M} \times \mathcal{K} \to \mathcal{C},$$

und die Entschlüsselungsfunktion

$$d\colon \mathcal{C} \times \mathcal{K} \to \mathcal{M}.$$

Für alle $M \in \mathcal{M}$ und $K \in \mathcal{K}$ muss die Bedingung

$$d(e(M,K),K) = M$$

gelten, damit die Entschlüsselung funktioniert.

5.2 One Time Pad

Das One Time Pad ist ein Beispiel für ein symmetrisches Kryptosystem. Bei diesem Verfahren wird ein zufälliger Schlüssel K gewählt, der genau die gleich Länge n wie die Nachricht M hat. Der Schlüssel wird bestimmt, indem n unabhängige zufällige Bits gewählt werden. Die Verschlüsselung und Entschlüsselung wird durch die bitweise Addition mod 2 der Bits von zwei Zeichenketten mit dem Operator $\oplus$ durchgeführt. Dieser wird auch als exklusives Oder bezeichnet.
Alice berechnet das Kryptogramm

$$C = e(M, K) = M \oplus K.$$

Wenn Bob auch den Schlüssel K kennt, kann er des Kryptogramm C mit

$$M = d(C, K) = C \oplus K$$

entschlüsseln. Das Verfahren funktioniert, da die folgende Bedingung eingehalten wird

$$d(e(M, K), K) = C \oplus K = (M \oplus K) \oplus K = M.$$

Der Name One Time Pad kommt daher, dass für jede versendete Nachricht zwischen Bob und Alice ein neuer Schlüssel gewählt wird. Dadurch ist dieses Verfahren sehr sicher (es hat sogar eine „perfekte Sicherheit“[12, S. 104f.], das bedeutet Eve kann nichts über die Nachricht aus dem Kryptogramm herausfinden).

Das One Time Pad hat aber auch mehrere Nachteile. Es müssen Schlüssel bestimmt werden, die genau so lang wie die Nachrichten sind. Das Bestimmen der Schlüssel ist also für lange Nachrichten sehr zeitaufwendig. Außerdem muss der Schlüssel vor dem Verfahren sicher von Alice zu Bob transportiert werden. Die Sicherheit ist also eher davon abhängig, wie Alice Bob den Schlüssel mitteilt und Bob diesen geheim hält, anstatt von der Sicherheit des eigentlichen Kryptosystems.

Die Nachteile können wir zum Teil beheben, indem wir den Schlüssel mit einem Pseudozufallszahlengenerator berechnen.

5.3 Symmetrisches Kryptosystem mit Pseudozufallszahlengenerator

Wir konstruieren nun ein symmetrisches Kryptosystem, dass auf dem One Time Pad basiert, mithilfe eines Pseudozufallszahlengenerators. Das Kryptogramm wird dann mit einer pseudozufälligen Zeichenkette anstatt einer echten zufälligen Zeichenkette berechnet.

Nehmen wir an, Alice und Bob wollen eine Nachricht austauschen und kennen den Pseudozufallszahlengenerator G.

- *Vorbereitung*: Alice wählt eine kurze zufällige Zeichenkette $x \in \{0,1\}^k$ und teilt diese geheim Bob mit.
- *Verschlüsselung*: Alice verschlüsselt die m-Bit lange Nachricht $M = (M_1, \ldots, M_m)$, indem sie mit G die pseudozufällige Zeichenkette

  $$G(x) = (B_1, \ldots, B_m)$$

  generiert und das Kryptogramm $C = G(x) \oplus M$ berechnet.
- *Entschlüsselung*: Bob erstellt die selbe pseudozufällige Zeichenkette $G(x)$ und berechnet die Nachricht mit $M = C \oplus G(x)$.

Auch bei diesem Kryptosystem muss Alice vor dem eigentlichen Verfahren Bob eine Zeichenkette schicken. Diese ist aber nicht der Schlüssel und für lange Nachrichten auch deutlicher kürzer als der Schlüssel, der beim One Time Pad Bob mitgeteilt werden muss.

Da $G(x)$ mit dem Pseudozufallszahlengenerator G aus x generiert wird, ist diese Zeichenkette pseudozufällig und nicht echt zufällig, wie bei einem echten One Time Pad. Nach unserer Definition für Pseudozufallszahlengeneratoren lässt sich $G(x)$ aber nicht von einem echten zufälligen Schlüssel unterscheiden. Wenn Eve relevante Informationen über den Schlüssel aus dem Kryptogramm erhalten kann, dann kann G kein Pseudozufallszahlengenerator sein (Satz 3.1).

6 Public-Key Verschlüsselung

Bei Public-Key Verschlüsselungsverfahren werden anstatt eines gemeinsamen Schlüssels, ein öffentlicher Schlüssel für die Verschlüsselung und ein privater Schlüssel für die Entschlüsselung verwendet. Die Grundlage für Public-Key Kryptosysteme sind Falltürfunktionen.

Definition (Falltürfunktion)**.** Eine Falltürfunktion f ist eine Einwegfunktion, die mithilfe einer bestimmten Information, dem Private-Key, effizient invertiert werden kann.

Wie Public-Key Verschlüsselungsverfahren funktionieren, zeigen wir am Beispiel von RSA.

6.1 RSA

RSA ist nach Ronal Linn Rivest, Adi Shamir und Leonard Max Adleman benannt, die das Verfahren 1978 veröffentlichten, und gilt als eins der bekanntesten Kryptosyteme. Dem Inhalt dieses Abschnitts liegen das Kapitel 7 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] und das originale Paper [11] zugrunde.

Das Verfahren funktioniert auf folgende Weise:

- *Vorbereitung*: Bob wählt zwei große Primzahlen $p \neq q$ und berechnet den öffentlichen Modulus $n = p \cdot q$. Dann berechnet Bob den öffentlichen Exponenten e, der teilerfremd zu $(p-1)\cdot(q-1)$ ist, mit $1 < e < (p-1)(p-q)$. Der öffentliche Modulus und der öffentliche Exponent bilden zusammen der öffentlichen Schlüssel (n, e), den Bob veröffentlicht. Sein privater Schlüssel ist eine Ganze Zahl $1 < d < (p-1)(q-1)$ mit der Eigenschaft
$$ed = 1 \bmod (p-1)(q-1)$$
- *Verschlüsselung*: Alice teilt die Nachricht M, die sie an Bob senden möchte, in eine Sequenz von Blöcken $M_1, \ldots, M_t$, wobei jeder Block M_i kleiner als n ist. Sie verschlüsselt diese Blöcke mit
$$C_i = M_i^e \bmod n,$$
und sendet Bob die verschlüsselten Blöcke.
- *Entschlüsselung*: Bob entschlüsselt die Blöcke mit seinem privaten Schlüssel d mit
$$M_i = C_i^d \bmod n.$$

Die Entschlüsselung funktioniert aus den folgenden Gründen. Das Kryptogramm wird berechnet als $C_i = M_i^e \bmod n$ und deswegen entspricht die Entschlüsselung

$$M_i = C_i^d = M_i^{ed} \bmod n.$$

Wir haben festgelegt, dass $ed = 1 \bmod (p-1)(q-1)$. Für eine Ganze Zahl t können wir auch schreiben $ed = 1 + t(p-1)(q-1)$. Dann können wir zeigen, dass die folgende Gleichung gilt,

$$M_i^{1+t(p-1)(q-1)} = M_i \bmod p.$$

Für $M_i = 0$ ist $M_i^{1+t(p-1)(q-1)}$ auch 0 und somit ist die Gleichung erfüllt. Für $M_i \neq 0$ können wir den kleinen fermatischen Satz (Grundlagen Satz 2.2) anwenden. Mit dem Satz folgt

$$M_i^{1+t(p-1)(q-1)} = M_i \cdot (M_i^{t(q-1)})^{(p-1)} = M_i \cdot 1 \bmod p.$$

Äquivalent können wir $M_i^{1+t(p-1)(q-1)} = M_i \bmod q$ zeigen. Mit dem Chinesischen Restsatz folgt dann $C_i^d = M_i^{1+t(p-1)(q-1)} = M_i \bmod n$, da $n = p \cdot q$. Die Entschlüsselung funktioniert also.

Die Funktion, die wir bei diesem Verfahren für die Verschlüsselungen von Nachrichten verwenden, bezeichnen wir von nun als RSA Funktion. Genauer ist es eine Gruppe von Funktionen $\mathrm{RSA}_{n,e}$, da sich die Funktion für verschieden Werte von n und e unterscheidet.

Definition ($\mathrm{RSA}_{n,e}$)**.** $\mathrm{RSA}_{n,e}$ ist eine Gruppe von Funktionen $\mathrm{RSA}_{n,e}$: mit $\mathrm{RSA}_{n,e}(x) = x^e \bmod n$, wobei $n = p \cdot q$ gilt und p, q zwei Primzahlen sind und e teilerfremd zu $(p-1)(q-1)$ ist.

Für diese Gruppe von Funktionen machen wir folgende Annahhme.

Definition (RSA Annahme)**.** Für jeden probababilistischen Polynomialzeitalgorithmus A gilt für ausreichend große k

$$\Pr[A(n, e, \mathrm{RSA}_{n,e}(x)) = x)] < neg(k),$$

für alle ganze Zahlen $n = pq$ mit verschiedenen k-Bit Primzahlen p, q, für alle e, die teilerfremd zu $(p-1)(q-1)$ sind und für alle zufälligen Bits, die A verwendet.

6.2 Verschlüsselung von einzelnen Bits

Nun wollen wir zeigen, wie wir mithilfe von Pseudozufallszahlen Verschlüsselungsverfahren erstellen können, die deutlich sichererer als das oben beschriebene RSA Verfahren sind. Dem Inhalt dieses und des folgenden Abschnitts liegt das Kapitel 10.5 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

Zuerst konstruieren wir ein Verschlüsselungsverfahren, mit dem nur Nachrichten, die aus einem Bit bestehen, versendet werden können.

Dabei wird ein großes Problem des oben beschriebenen RSA Verfahrens deutlich. Da der öffentliche und der private Schlüssel nicht jedes mal neu bestimmt werden, hat die gleiche Nachricht immer das gleiche Kryptogramm. Eve kann also erkennen, wenn eine Nachricht mehrmals geschickt wird und zwischen einzelnen Nachricht unterscheiden, indem sie die Kryptogramme vergleicht.

Wenn die Nachricht nur aus einem Bit besteht, kann Eve anhand des Kryptogramms zwischen allen möglichen Nachrichten unterscheiden.

Um dieses Problem zu lösen, erweitern wir das RSA Verfahren mit einer zufälligen Eingabe $x \in \mathbb{Z}_n^*$. Das x wird von Alice so gewählt, dass das Least Significant Bit von x der Nachricht $M \in \{0,1\}$ entspricht. Denn das Problem, das Least Significant Bit von x mit einer Erfolgswahrscheinlichkeit, die größer als $\frac{1}{2} + neg(n)$ (n ist der öffentliche Modulo) ist, aus dem Kryptogramm zu bestimmen, gilt als gleich schwer zu lösen, wie x aus dem Kryptogramm zu berechnen. Deswegen ist das Least Significant Bit ein Hardcore Prädikat von $\text{RSA}_{n,e}$ unter der RSA Annahme [4]. Da ungefähr die Hälfte der Werte in $\mathbb{Z}_n^*$ 0 als Least Significant Bit hat und die ander Hälfte 1, kann Alice relativ leicht ein passendes x finden. Der öffentliche Schlüssel (n, e) und der private Schlüssel d werden auf die gleiche Weise wie beim RSA Verfahren bestimmt.
Alice sendet Bob das Kryptogramm

$$C = x^e \bmod n.$$

Bob entschlüsselt das Kryptogramm mit seinem privaten Schlüssel und erhält

$$x = C^d \text{mod} n.$$

Von x nimmt Bob dann das Least Significant Bit und erhält die Nachricht M.

Nun hat jeweils ein x immer das gleiche Kryptogramm, aber nicht mehr die Nachricht selber. Deswegen kann Eve anhand des Kryptogramm nicht so einfach, wie bei RSA zwischen den Nachrichten unterscheiden.

Da das Least Significant Bit von x ein Hardcore Prädikat der $\text{RSA}_{n,e}$ Funktionen ist, kann ein möglicher Widersacher die Nachricht maximal mit einer Wahrscheinlichkeit, die kleiner als $\frac{1}{2} + neg(n)$ ist, richtig bestimmen.

Verschlüsselungsverfahren zum Verschlüsseln eines einzigen Bits definieren wir allgemein für alle Gruppen von Falltürfunktionen wie folgt.

Definition (Verschlüsselungsverfahren für einzelne Bits)**.** $\{f_i : D_i \to D_i\}_{i \in I}$ ist eine Gruppe von Falltürfunktionen mit den Hardcore Prädikaten $\{B_i\}_{i \in I}$.

- *Vorbereitung*: Bob wählt eine Schlüssellänge k und generiert einen privaten Schlüssel i und einen öffentlichen Schlüssel t_i. Er veröffentlicht i und hält t_i geheim.
- *Verschlüsselung*: Alice verschlüsselt ein einzelnes Bit $M \in \{0,1\}$. Dafür wählt sie ein zufälliges $x \in D_i$ mit $B_i(x) = M$ (Nach Satz 3.4 gibt $B_i(x)$ annähernd gleich

wahrscheinlich 0 oder 1 aus. Somit erfüllt $B_i(x) = M$ die Hälfte der Werte in D_i. So ein x kann also einfach gefunden werden).
Sie sendet Bob das Kryptogramm $C = f_i(x)$.

- *Entschlüsselung*: Bob benutzt seinen privaten Schlüssel t_i um x von $C = f_i(x)$ zu erhalten. Er berechnet dann die Nachricht $M = B_i(x)$.

6.3 Verschlüsselung von längeren Nachrichten

Das Verfahren für einzelne Bits erweitern wir nun so, dass wir auch längere Nachrichten verschlüsseln können. Dafür wenden wir das oben beschriebene Verfahren für jedes einzelne Bit einer Nachricht an.

Alice und Bob können Nachrichten, die länger als 1 Bit sind, mit dem folgenden Verfahren verschlüsseln.

Definition (Verschlüsselungsverfahren A)**.** $\{f_i : D_i \to D_i\}_{i \in I}$ ist eine Gruppe von Falltürfunktionen mit den Hardcore Prädikaten $\{B_i\}_{i \in I}$.

- *Vorbereitung*: Bob wählt eine Schlüssellänge k und generiert einen privaten Schlüssel i und öffentlichen Schlüssel t_i. Er veröffentlicht i und hält t_i geheim.
- *Verschlüsselung*: Alice verschlüsselt die m-Bit lange Nachricht $M \in \{0,1\}^m$. Dafür wählt sie zufällige $x_1, \ldots, x_m \in D_i$ mit $B_i(x_j) = M_j$ für $1 \leq j \leq m$. Sie sendet Bob das Kryptogramm $C = (f_i(x_1), f_i(x_2), \ldots, f_i(x_m))$.
- *Entschlüsselung*: Bob benutzt seinen privaten Schlüssel t_i, um jedes x_j aus $f_i(x_j)$ wiederherzustellen. Er berechnet dann die Nachricht $M = (B_i(x_1), B_i(x_2), \ldots, B_i(x_m))$.

Dieses Verfahren hat zwei große Nachteile. Das Kryptogramm ist bei längeren Nachrichten sehr lang: Wenn wir eine m-Bit lange Nachricht mit einem Hardcore Prädikat von $\mathrm{RSA}_{n,e}$ mit einem öffentlichen Modulo n der Länge k verschlüsseln, besteht das Kryptogramm aus m k-Bit langen Zeichenketten. Das gesamte Kryptogramm hat also die Länge mk.

Außerdem müssen für die Verschlüsselung m verschieden x_i gewählt werden, wodurch die Verschlüsselung sehr zeitaufwendig ist. Dieses Problem werden wir in Abschnitt 6.4 mit einem effizienteren Verfahren lösen.

Vorher wollen wir den Sicherheitsbegriff für Kryptosysteme genauer festlegen. Dafür definieren wir allgemein zwei Bedingungen, die ein Kryptosystem erfüllen muss, damit wir dieses als sicher bezeichnen können. Die erste Bedingung ist die polynomielle Ununterscheidbarkeit.

Definition (polynomielle Ununterscheidbarkeit.)**.** Nehmen wir an, Eve gibt Bob zwei Nachrichten M_1 und M_2, von denen er eine zu dem Kryptogramm C verschlüsselt. Ein Kryptosystem ist polynomiell ununterscheidbar, wenn Eve maximal mit einer Wahrscheinlichkeit, die vernachlässigbar größer als $\frac{1}{2}$ ist, bestimmen kann, von welcher der beiden Nachrichten C das Kryptogramm ist.

Um zu überprüfen, ob diese Bedingung gilt, benutzen wir den folgenden Test.

Test für polynomielle Ununterscheidbarkeit

1: Bob wählt eine Schlüssellänge k und generiert einen öffentlichen Schlüssel e und einen privaten Schlüssel d mit der gewählten Länge. Dann veröffentlicht er seinen öffentlichen Schlüssel e.
2: Eve erstellt zwei Nachrichten M_1, M_2 der Länge k in probabilistischer Polynomialzeit und gibt diese an Bob weiter.
3: Bob wählt zufällig eine der beiden Nachrichten $M \in \{M_1, M_2\}$ und verschlüsselt diese zum Kryptogramm $C = e(M)$.
4: Eve versucht zu bestimmen, welche der beiden Nachrichten von Bob verschlüsselt wurde.
5: Eve besteht den Test, wenn sie richtig bestimmt aus welcher Nachricht das Kryptogramm berechnet wurde.

Die Wahrscheinlichkeit, dass Eve diesen Test besteht, ist $\frac{1}{2}$, wenn sie die Antwort rät. Ein Kryptosystem ist genau dann polynomiell Ununterscheidbar, wenn Eve den Test maximal mit einer vernachlässigbar größeren Wahrscheinlichkeit als $\frac{1}{2}$ bestehen kann.

RSA ist nicht polynomiell Ununterscheidbar, da Eve selber die beiden Nachrichten mit dem öffentlichen Schlüssel verschlüsseln kann und eine Nachricht immer das gleiche Kryptogramm hat. Sie kann einfach ihre berechneten Kryptogramme mit dem Kryptogramm vergleichen, dass sie von Bob bekommt, und dadurch den Test bestehen.

Wir wollen nun zeigen, dass das Verschlüsselungsverfahren A polynomiell ununterscheidbar ist. Micali und Goldwasser lieferten dazu 1984 das folgende Ergebnis.

Satz 6.1 (Goldwasser und Micali [10])**.** *Jedes probabilistische Public-Key Kryptosystem ist polynomiell ununterscheidbar.*

Beweis. Wir werden hier nur eine Beweisskizze für diesen Satz machen. Der vollständige Beweis ist in [10]. Die Idee ist zu zeigen, dass Eve einen Polynomialzeit Algorithmus zur Bestimmung des Hardcore Prädikat erstellen kann, wenn sie zwischen den Verschlüsselungen von zwei bestimmten m-Bit Nachrichten unterscheiden kann.

Wir nehmen an, dass Eve zwischen den Verschlüsselungen der beiden Nachrichten M_1 und M_2 unterscheiden kann. Dann kann Eve eine Sequenz von Nachrichten erstellen, die mit M_1 starten und mit M_2 endet, indem sie von M_1 aus immer ein Bit invertiert bis sie die Nachricht M_2 erhält. Da Eve zwischen den Verschlüsselungen der Nachrichten M_1 und M_2 unterscheiden kann, gibt es in dieser Sequenz ein aufeinander folgendes Paar von Nachrichten, dessen Verschlüsselungen Eve unterscheiden kann. Das aufeinander folgende Paar von Nachrichten unterscheidet sich in nur einem Bit. Eve kann also auch zwischen den Verschlüsselungen von zwei Nachrichten der Länge m unterscheiden, die sich in in nur einem Bit unterscheiden.

Wenn Eve zwischen solchen Nachrichten unterscheiden kann, könnte sie einen Algorithmus mit polynomieller Laufzeit zur Bestimmen des Hardcore Prädikats erstellen, dessen Erfolgswahrscheinlichkeit nicht vernachlässigbar ist. □

Die zweite Bedingungen für sichere Kryptosysteme ist semantische Sicherheit. Informell bedeutet semantische Sicherheit, dass jede Information, die Eve aus dem Kryptogramm berechnen kann, genau so einfach ohne das Kryptogramm berechnet werden kann.

Wir sagen, dass Eve eine bestimmte Information erhalten kann, wenn es ihr möglich ist, eine Funktion $b\colon \mathcal{M} \to \{0,1\}$ zu lösen.

Definition (semantische Sicherheit)**.** Sei $b\colon \mathcal{M} \to \{0,1\}$ und die Nachricht M wird von Alice mit einem probabilistischem Polynomialzeit-Algorithmus erzeugt. Wir betrachten zwei verschiedene Fälle.

(1) *Ohne Kryptogramm*: Eve bekommt Bobs öffentlichen Schlüssel und die Information, dass die Nachricht die Länge k hat. Eve soll nun $b(M)$ bestimmen.

(2) *Mit Kryptogramm*: Eve bekommt Bobs öffentlichen Schlüssel und die Information, dass die Nachricht die Länge k hat. Außerdem geben wir ihr das Kryptogramm $C = e(M)$. Sie soll wieder $b(M)$ bestimmen.

Ein Kryptosystem ist genau dann semantisch sicher, wenn für jede Funktion $b\colon M \to \{0,1\}$ Eves Wahrscheinlichkeit $b(M)$ zu bestimmen in Fall (2) maximal unwesentlich größer als in Fall (1) ist.

Eine Funktion $b\colon \mathcal{M} \to \{0,1\}$ kann zum Beispiel wie folgt aussehen.

$$b(M) = \begin{cases} 1, & M \text{ enthält Bankdaten von Alice,} \\ 0, & \text{sonst.} \end{cases}$$

Die Funktion b ist also wie eine ja/nein Frage, ob eine bestimmte Information in der Nachricht enthalten ist. Eve kann somit nicht einmal herausfinden, welche Art von Information in der Nachricht enthalten ist, wenn das Kryptosystem semantisch sicher ist. Dabei ist es auch egal, ob Eve das Kryptogramm der Nachricht kennt oder nicht.

Goldwasser und Micali lieferten auch den folgenden wichtigen Zusammenhang zwischen den beiden Bedingungen für sichere Kryptosysteme.

Satz 6.2 (Goldwasser und Micali [10])**.** *Ein Kryptosystem ist genau dann semantisch sicher, wenn es polynomiell ununterscheidbar ist.*

Beweis. Siehe [10]. □

Wir müssen also nur eine der beiden Bedingungen für ein Public-Key Kryptosystem zeigen, dann ist dann Kryptosystem polynomiell ununterscheidbar und semantisch sicher.

6.4 Effiziente probabilistische Verschlüsselung

Wir wollen nun ein probabilistisches Verschlüsselungsverfahren konstruieren, wo bei der Verschlüsselung einer m-Bit langen Nachricht nur die Wahl eines einzigen $x \in D_i$ notwendig ist. Dem Inhalt dieses und des folgenden Abschnitts liegt das Kapitel 10.6 des Buches „Complexity and Cryptography - An Introduction“ von John Talbot und Dominic Welsh [12] zugrunde.

Das Verfahren ist eine Erweiterung des Verfahrens aus Abschnitt 5.3 zu einem Public-Key Verschlüsselungsverfahren, indem wir zur Erstellung des Pseudozufallszahlengenerators anstatt einer einfachen Einwegpermutation eine Gruppe von längenerhaltenden Falltürpermutationen $\{f_i \colon D_i \to D_i\}_{i \in I}$ mit den Hardcore Prädikaten $\{B_i\}_i \in I$ verwenden. Den Pseudozufallszahlengenerator erstellen wir wie in Satz 4.5. Das Verfahren funktioniert wie folgt.

Definition (effizientes probabilistisches Verschlüsselungsverfahren)**.** $\{f_i \colon D_i \to D_i\}_{i \in I}$ ist eine Gruppe von längenerhaltenden Falltürpermutationen mit den Hardcore Prädikaten $\{B_i\}_i \in I$.

- *Vorbereitung*: Bob entscheidet sich für eine Schlüssellänge und wählt einen öffentlichen Schlüssel i und einen privaten Schlüssel t_i mit der gewählten Länge. Er veröffentlicht den öffentlichen Schlüssel i.
- *Verschlüsselung*: Alice verschlüsselt eine Nachricht $M \in \{0,1\}^m$ wie folgt.
  - Sie wählt eine zufällige Saat $x \in D_i$.
  - Dann berechnet sie $f_i(x), {f_i}^2(x), \ldots, {f_i}^m(x)$.
  - Danach benutzt sie das Hardcore Prädikat B_i, um die pseudozufällige Zeichenkette
    $$P = (B_i(x), B_i(f_i(x)), \ldots, B_i({f_i}^{m-1}(x)))$$
    zu generieren.
  - Die Zeichenkette verwendet sie wie ein One-Time Pad und sendet Bob das Kryptogramm
    $$C = (P \oplus M, {f_i}^m(x)).$$
- *Entschlüsselung*: Wir schreiben $E = P \oplus M$ und $y = f_i^m(x)$. Bob entschlüsselt $C = (E, y)$ wie folgt.
  - Er benutzt seine privaten Schlüssel t_i um $x \in D_i$ so wiederherzustellen, dass $f_i^m(x) = y$ gilt.
  - Er kann nun P auf die gleiche Weise wie Alice generieren.
  - Dann berechnet er die Nachricht mit $M = E \oplus P$.

Satz 6.3. *Das effiziente probablilistische Verschlüsselungsverfahren ist polynomiell ununterscheidbar und somit auch semantisch sicher.*

Beweis. Die Idee ist, zu zeigen, dass Eve für einen beliebige Nachricht $M \in \{0,1\}^m$ und eine zufällige Zeichenkette die Verschlüsselungen nicht unterscheiden kann. Daraus können wir folgen, dass auch die Verschlüsselungen von zwei beliebigen Nachrichten für Eve ununterscheidbar sind.

Dafür müssen wir zuerst zeigen, dass die Zeichenkette $(P, {f_i}^m(x))$ pseudozufällig ist. Da wir für die Berechnung der Zeichenkette eine längenerhaltende Einwegfunktion f_i mit dem Hardcore Prädikat B_i benutzen, folgt mit Satz 4.4 und 4.5, dass

$$G_i(x) = (P, {f_i}^m(x)) = ((B_i(x), B_i(f_i(x)), \ldots, B_i({f_i}^{m-1}(x))), {f_i}^m(x))$$

ein Pseudozufallszahlengenerator ist. $(P, {f_i}^m(x))$ ist also pseudozufällig.
Nach Satz 3.1 muss ein Pseudozufallszahlengenerator alle statistischen Tests bestehen. Deswegen kann Eve zwischen $C = (P \oplus M, {f_i}^m(x))$ und $(R, {f_i}^m(x))$ nicht unterscheiden für eine Nachricht M und eine zufällige Zeichenkette R der Länge m, da sie ansonsten zwischen einer pseuodzufälligen Zeichenkette und einer echten zufälligen Zeichenkette unterscheiden könnte. Dadurch könnte sie einen statistischen Test T erstellen, den der Pseudozufallszahlengenerator G_i nicht besteht (wenn Eve die Eingabe von T von $(R, {f_i}^m(x))$ unterscheiden kann, gibt T eine 1 aus, sonst 0).

Wir können zwei beliebige Nachrichten M_1 und M_2 wählen. Dann sind die Kryptogramme$(P \oplus M_1, {f_i}^m(x))$ und $(P \oplus M_2, {f_i}^m(x))$ jeweils von $(R, {f_i}^m(x))$ ununterscheidbar. Daraus folgt, dass auch $(P \oplus M, {f_i}^m(x))$ und $(P \oplus M, {f_i}^m(x))$ ununterscheidbar sind. Das zeigt die polynomielle Ununterscheidbarkeit. □

Nun zeigen wir am Beispiel des Blum-Goldwasser Kryptosystems, wie ein effizientes probabilistisches Kryptosystem in der Praxis erstellt werden kann.

6.5 Das Blum-Goldwasser Kryptosystem

Das Blum-Goldwasser Kryptosystem verwendet eine Abwandlung der Gruppe von Funktionen RABIN_n zum berechnen von Pseduozufallszahlen.

Definition (RABIN_n)**.** RABIN_n ist eine Gruppen von Funktionen $\{\text{RABIN}_n \colon \mathbb{Z}_n \to \mathbb{Z}_n\}$ mit $\text{RABIN}_n(x) = x^2 \bmod n$, wobei $n = p \cdot q$ gilt und p, q verschiedene Primzahlen mit $p = q = 3 \bmod 4$ der Länge k sind.

Satz 6.4. *$RABIN_n$ ist unter der Faktorisierungsannahme eine Gruppe von Falltürfunktionen*

Beweis. Siehe [12, S. 161]. □

Wir wollen nun mit RABIN_n einen Pseudozufallszahlengenerator erstellen, wie in Satz 4.5. Die Bedingung dafür ist aber, dass wir eine Einwegpermutation benutzen. Wir müssen also RABIN_n so anpassen, dass wir eine Gruppe von Permutationen haben. Zur Erinnerung: Q_n ist die Gruppe der quadratischen Reste mod n.

Satz 6.5. *Wenn p,q Primzahlen der Form $p = q = 3$ mod 4 sind und $n = p \cdot q$, dann ist die Rabin-Williams Funktion $RW_n \colon Q_n \to Q_n, RW_n(x) = x^2$ mod n eine Permutation.*

Beweis. Die Idee für den Beweis ist, zu zeigen, dass für zwei Zahlen $x, y \in Q_n$ gilt, dass $x = y \bmod n$, wenn $\mathrm{RW}_n(x) = \mathrm{RW}_n(y)$. Das bedeutet, dass jeder Funktionswert unter RW_n ein eigenes Urbild hat. Da die Definitionsmenge und die Zielmenge der Funktion gleich sind, ist RW_n eine Permutation.
Die folgenden drei Informationen helfen uns bei dem Beweise.

(1) Es gilt $ab \in Q_p$ genau dann, wenn entweder $a \in Q_p$ und $b \in Q_p$ oder $a \notin Q_p$ und $b \notin Q_p$(Grundlagen Satz 2.3).

(2) Wenn $n = pq$ das Produkt von zwei verschiedenen Primzahlen ist, dann impliziert $b \in Q_n$, dass $b \in Q_p$ und $b \in Q_q$. Wir können $b \in Q_n$ auch schreiben als $b = k \cdot n + x^2$ für $x \in \mathbb{Z}_n^*$ und $k \in \mathbb{Z}$. Dann gilt $b = k \cdot n + x^2 = k \cdot p \cdot q + x^2$ und deswegen $b \in Q_p$ und $b \in Q_q$.

(3) Wenn p eine Primzahl der Form $p = 4k + 3$ ist und $a \in Q_p$, dann gilt $-a \notin Q_p$. Das folgt aus (1) und dem eulerschen Kriterium. Es gilt

$$(-1)^{(p-1)/2} = (-1)^{(4k+3-1)/2} = (-1)^{2k+1} = -1 \bmod p.$$

(-1) ist also kein quadratischer Rest mod p. Wenn $a \in Q_p$, dann ist das Produkt von a und (-1) auch kein quadratischer Rest wegen (1).

Jetzt können wir das eigentlich Ergebnis beweisen. Wir nehmen, an dass für $x, y \in Q_n$ gilt: $\mathrm{RW}_n(x) = \mathrm{RW}_n(y)$. Das können wir auch schreiben als $x^2 = y^2 \bmod n$ oder als $k \cdot n = x^2 - y^2 = (x + y) \cdot (x - y)$ für $k \in \mathbb{Z}$. Somit ist n ein Teiler von $(x + y) \cdot (x - y)$. Da $x, y \in Q_n$ wissen wir wegen (2), dass $x, y \in Q_p$ und $x, y \in Q_q$. Mit (3) folgt dann $-x, -y \notin Q_p$ und $-x, -y \notin Q_q$. Wegen (2) gilt dann auch $-x, -y \notin Q_n$.
Wir wissen also, dass $x \neq -y \bmod n$ gilt. Das können wir aufschreiben als $(x + y) \neq k \cdot n$. Es muss also entweder $(x - y) = 0$ gelten oder p ist ein Teiler von $(x - y)$ und q ist ein Teiler von $(x + y)$, damit die Gleichung $k \cdot n = (x + y) \cdot (x - y)$ erfüllt wird. Wenn ersteres gilt, ist $x = y$. Wenn letzteres gilt, dann ist $(x + y) = k \cdot q$ und somit $x = -y \bmod q$. Da $y \in Q_q$ und q die Form $q = 4k + 3$ hat, folgt $x \notin Q_q$ aus (3). Das ist ein Widerspruch dazu, dass $x \in Q_q$. Die Eingaben x und y der Funktion RW_n sind also gleich, wenn $\mathrm{RW}_n(x) = \mathrm{RW}_n(y)$. □

Lenore Blum, Manuel Blum und Michael Shub lieferten 1986 das folgende Ergebnis.

Satz 6.6 (Blum,Blum und Shub [6])**.** *Unter der Faktorisierungs Annahme ist die Gruppe der Rabin-Williams Funktionen $\{RW_n \colon Q_n \to Q_n\}_n$ eine Gruppe von Falltürpermutationen mit den Hardcore Prädikaten $B_n(x) =$ Least Significant Bit von x.*

Beweis. Siehe [6]. □

Mit der Gruppe der Rabin-Williams Funktionen $\{RW_n \colon Q_n \to Q_n\}_n$ mit den Hardcore Prädikaten $B_n(x)$ können wir dann einen Pseudozufallszahlengenerator erstellen. Dieser hat die folgende Funktionsweise:

Blum-Blum-Shub Generator

1: Wähle zwei verschiedene k-Bit Primzahlen p, q, der Form $p = q = 3 \bmod 4$ und berechne $n = pq$.
2: Wähle ein zufälliges $x \in \mathbb{Z}_n^*$.
3: Setze $x_0 = x^2 \bmod n$ und berechne $x_i = x_{i-1}^2 \bmod n$ für $i = 1$ bis $l(k)$.
4: Sei b_i das Least Significant Bit von x_i.
5: Ausgabe: $b_1, b_2, \ldots, b_{l(k)}$.

Nun können wir das Blum-Goldwasser Kryptosystem definieren, dass den Blum-Blum-Shub Generator zum Generieren einer pseudozufälligen Zeichenkette verwendet. Das Verfahren ist nach Manuel Blum und Shafi Goldwasser benannt.

Wir benutzen nicht alle Primzahlen der Form $p = 3 \bmod 4$, sondern nur die Teilmenge $\{p | p = 7 \bmod 8\}$, weil dadurch die Entschlüsselung vereinfacht wird [7].

Definition (Blum-Goldwasser Kryptosystem)**.** Das Blum-Goldwasser Kryptosystem ist ein probabilistisches Kryptosystem mit der folgenden Funktionsweise:

- *Vorbereitung*: Bob wählt eine Schlüssellänge k und wählt zwei zufällige k-Bit Primzahlen $p = 8i + 1$ und $q = 8j + 7$ mit $i, j \in \mathbb{Z}$. Dann erstellt er seinen öffentlichen Schlüssel $n = pq$ und veröffentlicht diesen. Sein privater Schlüssel ist das Paar (p, q).
- *Verschlüsselung*: Alice verschlüsselt die Nachricht $M \in \{0,1\}^m$ wie folgt
  - Zuerst wählt sie eine zufällige Saat $x_0 \in Q_n$, indem sie ein zufälliges $z \in \mathbb{Z}_n^*$ wählt und $x_0 = z^2 \bmod n$ berechnet.
  - Dann berechnet sie $x_i = x_{i-1}^2 \bmod n$ für $i = 1, \ldots, m$.
  - Danach benutzt sie das Hardcore Prädikat B_n (das Least Significant Bit von x), um die pseudozufällige Zeichenkette
    $$P = (B_n(x_0), B_n(x_1), \ldots, B_n(x_{m-1})).$$
    zu generieren.
  - Sie benutzt P als One-Time Pad und sendet Bob das Kryptogramm
    $$C = (P \oplus M, x_m).$$
- *Entschlüsselung*: Wir schreiben $E = P \oplus M$ und $y = x_m$. Bob entschlüsselt $C = (E, y)$ wie folgt.
  - Er benutzt seinen privaten Schlüssel (p, q), um x_0 von y wiederherzustellen.
  - Dann kann er P auf die gleiche Weise wie Alice konstruieren, da er die zufällige Saat x_0 und seinen öffentlichen Schlüssel n kennt.
  - Er berechnet die Nachricht als $M = E \oplus P$.

Der entscheide Schritt bei diesem Verfahren ist, x_0 aus y mit dem privaten Schlüssel wiederherzustellen. Da y mit einer Falltürpermutation berechnet wurde, kann Eve x_0 aus y ohne den privaten Schlüssel nicht effizient berechnet.

Wir zeigen nun, warum die Falltürfunktion genau so funktioniert und somit die Entschlüsselung ermöglicht.

Sei $b \in Q_n$, dann gilt auch $b \in Q_p$ und $b \in Q_q$(gleiche Begründung wie im Beweis für Satz 6.5). Nach dem eulerschen Kriterium ist dann $b^{(p-1)/2} = 1 \bmod p$. Wir habe die Form der Primzahl eingeschränkt auf $p = 8i + 7$. Deswegen können wir jetzt die Quadratwurzel von b bestimmen.

$$b = b * b^{(p-1)/2} = b \cdot b^{(8i+7-1)/2} = b^{1+4i+3} = (b^{2i+2})^2 \bmod p.$$

Somit ist $b^2i + 2 \bmod p$ die Quadratwurzel von b. Da $y = (x_0^2)^m$ gilt, müssen wir die 2^m-te Wurzel von y berechnen um $x_0 \bmod p$ zu entschlüsseln.

$$x = y^{(2i+2)^m} \bmod p.$$

Für q können wir x_0 ähnlich bestimmen: $x_0 = y^{(2j+2)^2} \bmod q$, wobei $q = 8j + 7$.

Mit dem chinesischen Restsatz und dem euklidischen Algorithmus, kann Bob $x_0 \bmod n$ wiederherstellen. Er benutzt zwei ganze Zahlen h, k, sodass $hp + kq = 1$, und berechnet dann

$$x_0 = kqy^{(2i+2)^m} + hpy^{(2j+2)^m} \bmod n.$$

Zuletzt erstellt Bob das Pad P auf die gleiche Weise wie Alice und berechnet die Nachricht $M = E \oplus P$.
Die Entschlüsselung des Blum-Goldwasser Kryptosystems funktioniert also.

7 Fazit

Das Ziel dieser Arbeit war es, zu zeigen, wie Pseudozufallszahlen erstellt werden können und welche Bedeutung diese für die Kryptographie haben.

Wir haben die Begriffe Pseudozufall und Bitgenerator eingeführt und Pseudozufallszahlengeneratoren über statistische Tests definiert. Mit dem Konzept der Hardcore Prädikate von Einwegfunktionen haben wir die Grundlagen für die Konstruktion von Pseudozufallszahlengeneratoren vorgestellt. Die Konstruktion haben wir allgemein und für zwei konkrete Beispiele mit den Einwegfunktionen dexp und RABIN gezeigt. Außerdem haben wir die Unterschiede von symmetrischen und asymmetrischen Kryptosystemen gezeigt und mit dem One-Time Pad ein Beispiel für ein symmetrisches Kryptosystem gegeben. Das One-Time Pad Verfahren haben wir mit dem Einsatz von Pseudozufallszahlen verbessert. Public-Key Kryptosysteme haben wir anhand vom nicht probabilistischen RSA erklärt. Wir haben probabilistische Kryptosystem für ein und mehrere Bit lange Nachrichten präsentiert und mit RSA verglichen. Wir haben außerdem mithilfe dieser Systeme polynomielle Ununterscheidbarkeit und semantische Sicherheit eingeführt, die die Sicherheit der Systeme genauer festlegen. Zum Schluss haben wir ein effizientes Verschlüsselungsverfahren mit dem Blum-Goldwasser Kryptosystem als Beispiel dafür gezeigt.

Wir haben festgestellt, dass Pseudozufallszahlen auf schweren mathematischen Problemen, wie zum Beispiel dem diskreter Logarithmus Problem, basieren. Die Pseudzufallszahlen sind also nur nicht von echten Zufallszahlen unterscheidbar, solange diese Probleme nicht effizient gelöst werden können.

Ein großer Vorteil von probabilistischen Kryptosystem gegenüber nicht probabilistischen Kryptosystemen ist, dass die verschlüsselten Nachrichten nicht immer das gleiche Kryptogramm haben. Gegenspieler können also Nachrichten nicht anhand der Kryptogramme unterscheiden und so gut wie keine Informationen aus dem Kryptogramm erhalten.

Außerdem kann mit Pseudozufallszahlen die aufwendige Wahl von echten Zufallszahlen erleichtert werden. Das haben wir am Beispiel vom One-Time Pad gesehen. Es werden nur kurze Zufallszahlen benötigt, die zu Pseudozufallszahlen der gewünschten Länge verlängert werden. Da Pseudozufallszahlen gewissermaßen nicht von echten Zufallszahlen unterschieden werden können, sind die Kryptosystem deswegen nicht weniger sicher.

Zusammenfassend bringen Pseudozufallszahlen viele Vorteile, vor allem im Bereich der Sicherheit, für die Kryptographie. Gleichzeitig bieten diese auch eine Angriffsstelle, wenn das zugrundeliegende Problem in der Zukunft gelöst werden kann.

Literatur

[1] *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977.

[2] *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 1982.

[3] Leonard M. Adleman, Kenneth L. Manders, and Gary L. Miller. On taking roots in finite fields. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* [1], pages 175–178.

[4] Werner Alexi, Benny Chor, Oded Goldreich, and Claus-Peter Schnorr. RSA and rabin functions: Certain parts are as hard as the whole. *SIAM J. Comput.*, 17(2):194–209, 1988.

[5] G. R. Blakley and David Chaum, editors. *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*. Springer, 1985.

[6] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.

[7] Manuel Blum and Shafi Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In Blakley and Chaum [5], pages 289–302.

[8] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984.

[9] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 25–32. ACM, 1989.

[10] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[11] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[12] John M. Talbot and Dominic J. A. Welsh. *Complexity and cryptography - an introduction*. Cambridge University Press, 2006.

[13] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982* [2], pages 80–91.

Selbstständigkeitserklärung

Hiermit erkläre ich, Felix Nils Ortmann, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als angegebene verwendet habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Werken entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Hannover, den 21. Oktober 2019

_____

Felix Nils Ortmann